Игорь Одинцов

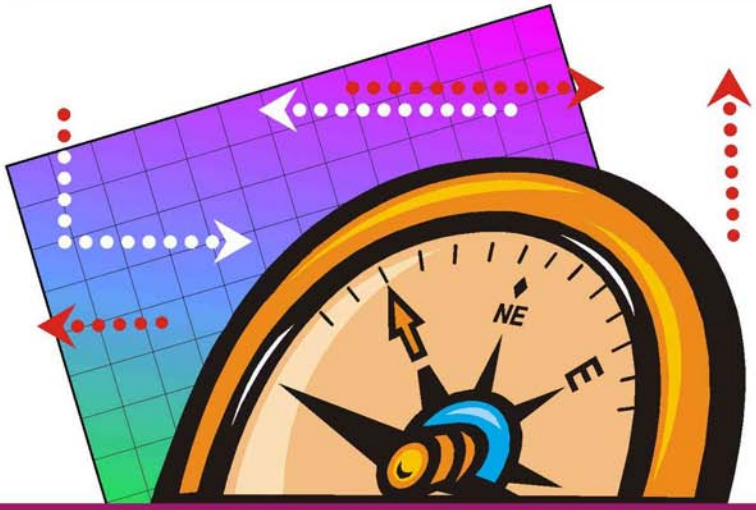
bhv

Профессиональное программирование

Системный подход

2-е издание

- Методологии и технологии программирования
- Языки и системы программирования
- Аппаратная и операционная платформы
- Управление программным проектом
- Социальные вопросы и профессиональная карьера



МАСТЕР

Игорь Одинцов

Профессиональное программирование

Системный подход

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068
ББК 32.973-018
О-42

Одинцов И. О.

О-42 Профессиональное программирование. Системный подход.— 2-е изд. перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 624 с.: ил.

ISBN 5-94157-457-6

Подробно рассматриваются методологии, технологии, языки и системы программирования, аппаратная и операционная платформы, как давно известные, так и новейшие. Представлены различные их классификации. Описывается управление программным проектом и его особенности. Обсуждаются планирование, мотивация и реализация профессиональной карьеры в программировании и другие важные социальные вопросы информатики. Предлагается большой список литературы, включающий основные работы по каждой из исследуемых областей. Во втором, существенно переработанном и дополненном издании, отражены изменения последних лет в области информационных систем и дана оценка текущему состоянию дел в анализируемых областях.

Для начинающих и профессиональных программистов, менеджеров проектов

УДК 681.3.068
ББК 32.973-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Нина Седых*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Татьяна Звертановская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 05.05.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 50,31.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-457-6

Одинцов И. О., 2004
Оформление, издательство "БХВ-Петербург", 2004

Содержание

Предисловие 11

Цель книги 11
Для кого предназначена эта книга 13
Организация книги 13
Профессиональные рамки 14
Основные направления профессионального программирования 15
Рекомендации по порядку чтения глав 18
Принятые в книге обозначения 18
Благодарности 19

Предисловие ко 2-му изданию 21

Основные изменения в книге и в мире 21
Книга как отправная точка 22
Благодарности 23

Глава 1. Профессиональное программирование 25

1.1. Профессионализм в программировании 25
1.1.1. Функции, черты и особенности мышления профессиональных программистов 27
1.2. Программирование. Что это такое? 31
1.3. Программирование. Чему учиться? 34
1.3.1. Подход на основе рекомендаций ACM/IEEE 35
1.3.2. Российские программы по информатике 37
1.3.3. Точка зрения опытных программистов 39
1.3.4. Подход, предлагаемый данной книгой 41
1.4. Программирование. Как учиться? 43
1.4.1. Подход на основе рекомендаций ACM/IEEE 43
1.4.2. Рекомендации российских специалистов 45
1.4.3. Авторские подходы 46
1.4.4. Отступление "О роли наставника в процессе обучения" 48
1.5. Программирование. Искусство, наука, ремесло или технология? 49

1.6. Кризис программирования 53
1.7. Основные понятия профессионального программирования 55
1.7.1. Отступление "о понятиях, определениях и классификациях" 55
1.7.2. Программа и программный продукт 57
1.7.3. Алгоритмы 58
1.7.4. Модели и моделирование 60
1.7.5. Жизненный цикл программного обеспечения 61
1.7.6. Проект и команда 62
1.7.7. Платформы 67
1.8. Основная задача профессионального программирования 69
Литература 70

Глава 2. Методологии программирования 73

2.1. Введение в методологии программирования 73
2.1.1. Основные понятия и определения 73
2.1.2. История и эволюция 75
2.1.3. Классификации методологий 77
2.2. Ядра методологий 79
2.2.1. Методология императивного программирования 79
2.2.2. Методология объектно-ориентированного программирования 83
2.2.3. Методология функционального программирования 87
2.2.4. Методология логического программирования 91
2.2.5. Методология программирования в ограничениях 94
2.3. Топологическая специфика методологий 97
2.3.1. Методология структурного императивного программирования 97
2.4. Реализационная специфика методологий 99
2.4.1. Методология императивного параллельного программирования 99
2.4.2. Методология логического параллельного программирования 103
2.5. Заключение 105
2.5.1. Выводы и рекомендации 105
2.5.2. Проблемы и перспективы развития 105
2.5.3. Рекомендации по литературе 107
2.5.4. Вопросы и задания 108
Литература 108

Глава 3. Технологии программирования 111

3.1. Введение в технологии программирования 111
3.1.1. Основные понятия и определения 111
3.1.2. История и эволюция 112
3.1.3. Классификации 113
3.2. Классические технологические процессы 117
3.2.1. Возникновение и исследование идеи 117

3.2.2. Управление ... 122
3.2.3. Анализ требований и проектирование ... 136
3.2.4. Программирование (реализация) ... 153
3.2.5. Тестирование и отладка ... 161
3.2.6. Ввод программы в действие ... 166
3.2.7. Эксплуатация и сопровождение ... 167
3.2.8. Завершение эксплуатации ... 170
3.3. Стандартные технологические процессы ... 171
3.3.1. Основные процессы ... 171
3.3.2. Вспомогательные процессы ... 173
3.3.3. Организационные процессы ... 175
3.3.4. Взаимосвязь между процессами ... 176
3.4. Основные стадии технологических подходов ... 177
3.4.1. Фазы как крупные временные рамки ... 177
3.4.2. Стадии как отражение классических процессов ... 177
3.4.3. Вариант подробного разбиения на стадии ... 178
3.4.4. Контрольные точки ... 180
3.5. Основные технологические подходы ... 181
3.5.1. Ранние технологические подходы ... 181
3.5.2. Каскадные технологические подходы ... 181
3.5.3. Каркасные технологические подходы ... 185
3.5.4. Генетические технологические подходы ... 190
3.5.5. Подходы на основе формальных преобразований ... 192
3.5.6. Группа ранних подходов быстрой разработки ... 196
3.5.7. Адаптивные технологические подходы ... 198
3.5.8. Подходы исследовательского программирования ... 203
3.6. Технологии коллективной разработки ... 205
3.6.1. Авторская разработка ... 205
3.6.2. Коллективная разработка ... 206
3.6.3. Общинная модель разработки ... 213
3.6.4. Отступление "об оффшорном программировании" ... 214
3.7. Качество программного обеспечения ... 216
3.7.1. Подходы к качеству программного обеспечения ... 217
3.7.2. Характеристики качества программного обеспечения ... 218
3.7.3. Оценка качества процесса разработки ... 220
3.7.4. "Достаточно хорошее" программное обеспечение ... 223
3.7.5. Стандартизация информационных технологий ... 224
3.8. Заключение ... 226
3.8.1. Выводы и рекомендации ... 226
3.8.2. Проблемы и перспективы развития ... 229
3.8.3. Рекомендации по литературе ... 230
3.8.4. Вопросы и задания ... 231
Литература ... 233

Глава 4. Языки программирования.....237

4.1. Введение в языки программирования 237
4.1.1. Основные понятия и определения.....238
4.1.2. История и эволюция.....245
4.1.3. Классификация языков.....247
4.2. Языки, поддерживающие классические технологические процессы..249
4.2.1. Возникновение и исследование идеи.....249
4.2.2. Управление 249
4.2.3. Анализ требований и проектирование 251
4.2.4. Программирование (реализация).....251
4.2.5. Тестирование и отладка 251
4.2.6. Ввод в действие 251
4.2.7. Эксплуатация и сопровождение.....251
4.2.8. Завершение эксплуатации.....252
4.3. Языковые абстракции.....252
4.3.1. Отступление "об абстрагировании".....252
4.3.2. Абстракция данных.....253
4.3.3. Абстракция управления.....257
4.3.4. Абстракция модульности.....264
4.4. Языки моделирования 269
4.4.1. Моделирование на основе структурной методологии 269
4.4.2. Моделирование на основе объектно-ориентированной методологии.....271
4.4.3. Языки моделирования данных.....277
4.4.4. Языки моделирования знаний.....279
4.5. Языки программирования высокого уровня.....282
4.5.1. Обзор языков, принадлежащих к семействам 282
4.5.2. Обзор языков, ориентированных на предметную область.....288
4.6. Формальные языки 291
4.6.1. Иерархия грамматик.....292
4.6.2. Техники распознавания (разбора).....293
4.6.3. Образное сравнение типов грамматик 294
4.6.4. Метатрансляция 294
4.6.5. Язык спецификации грамматик.....297
4.7. Естественные языки.....298
4.7.1. Особенности естественных языков и культурных сред.....298
4.7.2. Семантический анализ естественных языков.....300
4.7.3. Интернационализация и локализация программных продуктов ...302
4.7.4. Отступление "о придании тексту эмоций".....304
4.8. Заключение 306
4.8.1. Выводы и рекомендации.....306
4.8.2. Проблемы и перспективы развития.....307
4.8.3. Рекомендации по литературе.....308
4.8.4. Вопросы и задания 309
Литература.....311

Глава 5. Системы программирования 315

5.1. Введение в системы программирования 315
5.1.1. Основные понятия и определения 315
5.1.2. История и эволюция 317
5.1.3. Классификация 318
5.2. Процесс-ориентированный инструментарий 320
5.2.1. Возникновение и исследование идеи 320
5.2.2. Управление 321
5.2.3. Анализ требований и проектирование 324
5.2.4. Программирование (реализация) 326
5.2.5. Тестирование и отладка 338
5.2.6. Ввод в действие 343
5.2.7. Сопровождение 343
5.2.8. Завершение эксплуатации 345
5.3. Универсальный инструментарий 345
5.3.1. Инструменты работы с текстом 345
5.3.2. Системы документирования 351
5.3.3. Системы разработки интерфейсов 353
5.3.4. Системы управления базами данных 353
5.3.5. Системы управления базами знаний и экспертные системы 354
5.3.6. Электронные библиотеки и инструментарий Интернета 358
5.4. Инструментарий поддержки процессов некоторых технологических подходов 368
5.4.1. Системы верификации программ 368
5.4.2. Системы формального преобразования программ 368
5.4.3. Средства сборочного программирования 368
5.5. Инструментальные системы 369
5.5.1. Инструментальные среды программирования 369
5.5.2. Средства автоматизации разработки программ (CASE-средства) 370
5.5.3. Интегрированные среды 371
5.5.4. Репозитории проекта 372
5.6. Средства поддержки коллективной разработки 376
5.6.1. Системы разделения файлов 376
5.6.2. Системы поддержки работы виртуальных групп 380
5.7. Естественно-языковый интерфейс 382
5.7.1. Диалоговые системы 382
5.7.2. Вопросно-ответные системы 383
5.7.3. Автоматизированные обучающие системы и системы контроля знаний 384
5.7.4. Системы искусственного интеллекта 385
5.8. Заключение 386

5.8.1. Выводы и рекомендации ... 386
5.8.2. Проблемы и перспективы развития ... 389
5.8.3. Рекомендации по литературе ... 391
5.8.4. Вопросы и задания ... 392
Литература ... 393

Глава 6. Архитектурная платформа ... 397

6.1. Основы архитектуры ЭВМ ... 397
6.1.1. Основные понятия и определения ... 397
6.1.2. История и эволюция компьютерных архитектур ... 398
6.1.3. Классификация вычислительных систем ... 402
6.2. Основные архитектуры набора команд ... 406
6.2.1. Классические архитектуры ... 406
6.2.2. Архитектуры CISC, RISC и VLIW ... 409
6.2.3. Вычисления ... 411
6.2.4. Архитектура ЭВМ и языки программирования ... 414
6.2.5. Оценка производительности вычислительных систем ... 414
6.3. Организация вычислительной системы ... 416
6.3.1. Процессор ... 416
6.3.2. Память ... 416
6.3.3. Шина ... 417
6.3.4. Устройства ввода и вывода данных ... 418
6.3.5. Функциональное описание ... 418
6.4. Особенности SPARC-архитектуры ... 419
6.4.1. История создания SPARC-архитектуры ... 419
6.4.2. Основные понятия SPARC-архитектуры ... 420
6.4.3. Основной набор команд языка Assembler ... 423
6.4.4. Примеры программ на языке Assembler ... 429
6.4.5. Организация SPARC-архитектуры ... 432
6.5. Параллельные и распределенные архитектуры ... 434
6.5.1. Основные классы параллельных архитектур ... 434
6.5.2. Коммутаторы вычислительных систем ... 435
6.5.3. Некоторые классификации параллельных архитектур ... 437
6.6. Архитектура компьютерных сетей ... 439
6.6.1. Классификация сетей и сетевые топологии ... 439
6.6.2. Стандарты в области сетей IEEE 802 ... 441
6.6.3. Аппаратная поддержка локальных сетей ... 442
6.6.4. Глобальная сеть Интернет ... 444
6.7. Заключение ... 447
6.7.1. Выводы и рекомендации ... 447
6.7.2. Проблемы и перспективы развития ... 449
6.7.3. Рекомендации по литературе ... 451
6.7.4. Вопросы и задания ... 452
Литература ... 453

Глава 7. Операционная платформа 455

7.1. Введение в операционные системы 455
7.1.1. Основные понятия и определения 455
7.1.2. История и эволюция операционных систем 457
7.1.3. Классификация операционных систем 460
7.2. Процессы 465
7.2.1. Процессы и потоки (нити) управления 465
7.2.2. Коммуникация и синхронизация процессов в централизованных архитектурах 474
7.2.3. Коммуникация процессов в сетях 491
7.2.4. Синхронизация процессов в распределенных системах 505
7.3. Память 514
7.3.1. Основная память 514
7.3.2. Внешняя память 521
7.4. Драйверы устройств 526
7.5. Заключение 527
7.5.1. Выводы и рекомендации 527
7.5.2. Проблемы и перспективы развития 529
7.5.3. Рекомендации по литературе 531
7.5.4. Вопросы и задания 532
Литература 534

Глава 8. Элементы менеджмента и социальные вопросы информатики 537

8.1. Введение в элементы менеджмента и социальные вопросы информатики 537
8.2. Набор новых сотрудников 540
8.2.1. Привлечение кандидатов 541
8.2.2. Анализ краткой биографии (резюме) 542
8.2.3. Собеседование 547
8.2.4. Испытательный срок 564
8.3. Карьера в области информационных систем 565
8.3.1. Карьера и ее планирование 565
8.3.2. Совершенствование мастерства 569
8.3.3. Олимпиадное (соревновательное) программирование 571
8.3.4. Философия и стратегия программистских коллективов и организаций 578
8.3.5. Формы власти и стили руководства 579
8.3.6. Договорная основа трудовых отношений 580
8.3.7. Заработная плата 581
8.4. Почему некоторые программисты на порядок полезнее других? 584
8.4.1. Насколько корректна постановка проблемы? 585

8.4.2. Характеристики хорошего программиста.....587
8.4.3. Можно ли стать полезнее?.....591
8.5. Этические ответственности.....592
8.5.1. Этика в работе программиста.....592
8.5.2. "Феномен науки" В. Ф. Турчина.....595
8.5.3. Отступление "о свободе и стоимости программного обеспечения".....596
8.6. Заключение.....599
8.6.1. Выводы и рекомендации.....599
8.6.2. Рекомендации по литературе.....601
8.6.3. Вопросы и задания.....602
Литература.....603

Предметный указатель.....605

Предисловие

Цель книги

Знать цель, к которой мы стремимся — это рассудительность;
достигнуть этой цели — верность взгляда;
остановиться на ней — сила;
попасть далее цели — дерзость.

Ш. Дюкло

В информационном океане давно появился архипелаг "Программирование". К первому однобитному островку, вынырнувшему из пучин, стремительно начали пристраиваться другие. Здесь есть и большие надежные острова, например, операционной системы Unix, языка программирования Pascal, объектно-ориентированной методологии. Есть молодые с бурной вулканической деятельностью и неустойчивой почвой. Есть малые пятачки terra incognita. Некоторые острова, исчерпав себя, распались сами, и о них забыли. Другие, имеющие прочную основу, укрепились надолго.

Если Вы программист-"капитан", программист-"бывалый моряк" или пока просто программист-"юнга", то корабль с компасом и цель путешествия у Вас уже есть. Чтобы хорошо ориентироваться среди островов этого архипелага, нужны еще карта и хороший лоцман. Книга, которую Вы держите в руках, — именно то, что Вам необходимо. Она дает Вам возможность проложить и в дальнейшем уточнять курс движения. Маяки, установленные для Вас, позволят Вам не заблудиться в тумане и обнаружить новые острова.

При ее написании автор ставил перед собой две цели.

- Познакомить Вас с современным состоянием развития основных областей программирования. С помощью этой книги Вы узнаете или пополните свои знания о методологиях, технологиях, языках и системах программирования, аппаратной и операционной платформах, как хорошо себя зарекомендовавших и устоявшихся, так и новейших.
- Предложить Вам систематизацию и классификацию этих областей.

В соответствии с замыслом в книге излагается много фактов и концепций, являющихся весьма разнородными. Ее содержание сложилось в результате анализа большого количества научных и научно-популярных источников и с учетом профессионального опыта автора.

При создании этой книги автор следовал следующим принципам.

- **Взаимосвязанный, системный подход к программированию.** Жизненный цикл программы практически в каждый момент времени имеет отношение к некоторому технологическому процессу, некоторому языку и некоторому инструменту системы программирования. Поэтому нам необходимо не только знание каждого из этих направлений, но и понимание теснейших внутренних и внешних связей и зависимостей между ними. А определяющую роль в выборе языков и систем (и, во многом, технологий) играет методология программирования.
- **Изложение программирования в обобщенном и систематизированном виде.** С одной стороны, здесь мы будем избегать излишнего теоретизирования. С другой стороны, мы не станем углубляться в технические детали, которые лучше изучать с помощью специализированной программной документации. Говоря проще, в большинстве случаев Вы не найдете здесь утверждений типа "переместите указатель на иконку и щелкните кнопкой мыши".

Давайте представим множество знаний в некоторой предметной области в виде дерева. Ствол дерева олицетворяет "практически вечные" знания (их немного), ветви — "долговременные", а листья — "мимолетные" (их подавляющее большинство). Для предметной области программирования эти три группы знаний выглядят так:

- фундаментальные знания, математические основы программирования (такие знания обычно остаются актуальными как минимум на протяжении человеческой жизни);
- знание методологий программирования, групп языков, классов систем и технологий, типов архитектурных и операционных платформ (время устаревания таких знаний около 10—15 лет);
- знание технических особенностей конкретной аппаратуры или конкретных программных средств (устаревание около 3 лет).

Основное внимание в этой книге уделяется второй группе знаний — "золотой середине". Иногда, в виде исключения, мы дадим Вам возможность явно полюбоваться красивым (но, в большинстве случаев, недолговечным) "листиком".

Отметим, что из-за невозможности "объять необъятное", данная книга во многих местах имеет краткий (энциклопедический) характер. Эта особенность компенсируется обилием ссылок на дополнительную литературу и страницы в сети Интернет. Эти ссылки позволят Вам самостоятельно вырастить многочисленные "листья" на уже известных Вам "ветвях".

Для кого предназначена эта книга

Из всех связей, возникающих между людьми, самой непостоянной, самой запутанной и изменчивой является связь между писателем и читателем.

А. Шефтсбери

Книга будет полезна всем тем, кто занимается или собирается заниматься разработкой программного обеспечения.

- Профессионалы, уже имеющие большой опыт разработки программ, могут использовать данную книгу для систематизации своих знаний, а также в качестве справочника по основным разделам программирования. Книга поможет дополнить или уточнить имеющиеся у них данные, а также получить ссылки на подробные источники информации.
- Менеджеры проектов могут использовать информацию о процессе управления разработкой программного обеспечения.
- Начинающие программисты могут получить из книги представление об основных областях профессионального программирования, связанных с разработкой программ. Эта книга может являться для них стартовой площадкой, "печкой", от которой можно и нужно двигаться дальше, переходя к книгам по конкретным языкам и системам, а также к постоянному и систематическому практическому программированию.
- Студенты могут использовать отдельные главы книги в качестве учебного пособия. Особенно полезна книга будет студентам, обучающимся по специальностям 220400 "Программное обеспечение вычислительной техники и автоматизированных систем", 351400 "Прикладная информатика по областям", 351500 "Математическое обеспечение и администрирование информационных систем" и близким к ним.
- Преподаватели могут использовать книгу в целом и отдельные ее главы в качестве основы для построения развернутых лекционных курсов по этой тематике.

Организация книги

То, что я понял, прекрасно, из этого я заключаю, что остальное, чего я не понял, тоже прекрасно.

Сократ (по поводу неясностей у Гераклита)

Книга состоит из восьми глав. Первая и восьмая — стартовая и заключительная — представляют практические знания и рекомендации в области профессионального программирования. Главы со второй по седьмую — это

внутренние главы, посвященные важнейшим направлениям профессионального программирования.

Профессиональные рамки

Глава 1. Профессиональное программирование

В этой главе рассматриваются вопросы, над которыми рано или поздно задумываются все программисты.

- Каковы функции, черты и особенности мышления профессиональных программистов?
- Что должны знать и уметь профессиональные программисты?
- Существуют ли стандарты для обучения программированию?
- Чем является программирование: искусством, наукой, ремеслом или технологией?
- Почему многие программные проекты выполняются с отставанием от графика, с превышением сметы расходов, при этом качество продукта не устраивает пользователя?

Завершающая часть главы посвящена основным понятиям и определениям из области профессионального программирования.

Глава 8. Элементы менеджмента и социальные вопросы информатики

Разделы главы объединены понятием "человеческий фактор". Эта глава в одинаковой степени полезна и программистам, желающим устроиться на работу, и руководителям проектов, которые принимают их на работу. Мы рассмотрим уровни карьеры, вопросы планирования карьеры и примеры составления некоторых важных документов. Далее мы перечислим основные практические вопросы главы.

- Как сделать карьеру в области информационных систем? В чем особенности национальной карьеры?
- Какими должны быть первые шаги в поиске работы? Как правильно составить резюме? Как проводится собеседование и как к нему подготовиться?
- Почему некоторые программисты на порядок полезнее других?
- Что такое олимпиадное (соревновательное) программирование? Какая от него польза и связано ли оно с реальным программированием?
- Каковы этические проблемы, возникающие при разработке программного обеспечения? Как их можно решать с этической и технической точек зрения?

Основные направления профессионального программирования

Прежде чем ознакомить вас с кратким изложением основных вопросов, затронутых в главах 2—7, определим главные направления. Основные области в профессиональном программировании представим тремя взаимосвязанными группами. Каждая группа — это объединение двух достаточно близких направлений.

Направляющая группа

Это группа содержит два направления, с которыми следует определиться перед началом работы с программным проектом. Выбранные методы и подходы определят основную идеологию и принципы проекта.

Глава 2. Методологии программирования

В этой главе рассказывается об основных методологиях программирования — совокупности методов, применяемых в жизненном цикле программного обеспечения и объединенных общим философским подходом. Основные вопросы главы сформулируем следующим образом.

- Зачем нужны определяющие методологии в программировании?
- В чем заключается поддержка методологии методами и концепциями?
- Каковы вычислительные модели методологий?
- На какой класс практических задач ориентирована каждая из методологий?
- Какова языковая поддержка методологий?
- Каково происхождение методологий?

Глава 3. Технологии программирования

Прочитав эту главу, вы получите ответы на вопросы, приведенные ниже.

- Что такое жизненный цикл программ и каковы основные технологические подходы к его ведению?
- Что такое классические и стандартные технологические процессы?
- Как формируются технологические стадии и в чем заключается порядок их прохождения?
- Что представляют собой технологии коллективной разработки программ?
- Как добиться создания качественного программного обеспечения?

Инструментальная группа

Эта группа содержит два инструментальных направления: языковую и системную поддержку проекта.

Глава 4. Языки программирования

Мы рассмотрим языки, участвующие в создании программы, поддерживающие основные технологические процессы. Для двух основных языковых групп — языков моделирования и языков программирования высокого уровня — можно выделить черты, о которых также пойдет речь в этой главе.

- Язык обладает характеристиками.
- Язык содержит абстракции.
- Яык поддерживает методологию программирования.
- Яык принадлежит к семейству языков.
- Яык может быть специализированным для некоторой предметной области.

Завершается глава кратким обсуждением формальных и естественных языков.

Глава 5. Системы программирования

В этой главе мы подробно поговорим о нескольких группах инструментов, которые относятся к системам программирования, а именно:

- инструментарий поддержки технологических процессов. Мы выделим инструменты, ориентированные и применяемые только в одном процессе, и универсальные (независимые от процесса) инструменты;
- инструментальные системы (инструментальные среды программирования, средства автоматизации разработки программ, интегрированные среды и репозитории проекта);
- инструментарий поддержки коллективной разработки.

Глава также включает обзор систем поддержки естественно-языкового интерфейса.

Базисная группа

Эта группа включает два направления, представляющих собой основные платформы, на которых базируется программный проект.

Глава 6. Архитектурная платформа

Вся эта глава посвящена компьютерным архитектурам. Рассматриваются три основные группы вопросов.

- Оновные архитектуры набора команд — классические, современные и нестандартные.

- Оганизация вычислительной системы — процессор, память, шина, функциональное описание.
- Праллельные и распределенные архитектуры. В том числе сети ЭВМ, как частный случай распределенной архитектуры.

Глава 7. Операционная платформа

Изучение механизма и структуры операционных систем необходимо по следующим причинам:

- основные идеи, концепции и алгоритмы, лежащие в основе операционных систем, применимы ко многим другим областям программирования, и особенно к системному программированию;
- операционная система — большая и очень сложная программа, на примере которой можно изучать вопросы создания сложных программных продуктов;
- такие популярные программные продукты, как системы управления базами данных, могут рассматриваться как надстройки над операционными системами.

Каждая из глав 2—7 начинается тремя разделами, дающими подробное введение в изучаемую область:

- основные понятия и определения;
- история и эволюция;
- классификации.

Завершают каждую из этих глав четыре раздела, подводящих итоги рассмотренному материалу:

- выводы и рекомендации;
- проблемы и перспективы развития;
- рекомендации по литературе;
- вопросы и задания.

Литература

Чтение сделало Дон-Кихота рыцарем, а вера в прочитанное сделала его сумасшедшим.

Джордж Бернард Шоу

Книга содержит список литературы, охватывающий основные работы в исследуемых областях, изданные в нашей стране и за рубежом. Несмотря на то, что мы стремимся включить в книгу самую актуальную информацию, рано или поздно она устаревает. Эффективно бороться с этим можно только с помощью Интернета. Если Вы еще не знаете как, то книга научит Вас методам профессионального поиска.

Рекомендации по порядку чтения глав

Природа подобна женщине, которая... показывая
из-под своих нарядов то одну часть своего тела, то другую,
подает своим настойчивым поклонникам
некоторую надежду узнать ее когда-нибудь всю.

Дени Дидро

Мы предлагаем три основных варианта прочтения этой книги.

- С последовательным обращением. Этот способ классический, когда главы читают в порядке их расположения. Этот вариант предлагается в первую очередь начинающим программистам.
- С параллельным обращением. Рекомендуется желающим систематизировать свои знания. Он требует параллельного чтения нескольких глав, поскольку некоторые изучаемые в книге вопросы образуют тройственную взаимосвязь, которую надо отслеживать. Так, в третьей главе может излагаться некоторый технологический процесс, в четвертой — языки, а в пятой — системы, его поддерживающие.
- С произвольным обращением. Третий способ — обращение к главам по мере необходимости или интереса читателя к изложенным вопросам. Опытные программисты могут использовать книгу как справочник, просматривая материал по интересующим их методологиям и технологиям, языкам и системам, аппаратной и операционной платформам.

Принятые в книге обозначения

Уже сам знак параграфа выглядит как орудие пытки.

Ежи Лец

Для облегчения работы с текстом в книге используются следующие соглашения:

- коды программ и фрагменты документов выделены специальным шрифтом (`Courier`);
- базовые понятия и определения выделены *курсивом* при их первом появлении в тексте;
- замечания к основному тексту выделены рубленым шрифтом;
- для ссылок в тексте на литературу на бумажном носителе используются квадратные скобки (например, [Лавров 2001]). Списки литературы приводятся в конце каждой главы;

- ссылки на ресурсы Интернета (адреса сайтов) даются непосредственно в тексте книги;
- в книге применяется система нумерации, достигающая четырех уровней вложенности:
  - номера глав состоят из одной цифры, ссылка на главу производится, например, так: *гл. 1*;
  - номера разделов состоят из двух, трех и четырех цифр, ссылки на разделы осуществляются, например, так: *разд. 1.2*, *разд. 1.2.3*, *разд. 1.2.3.4*;
- названия языков и систем программирования, операционных систем и т. п. будут приводиться на русском языке, если они созданы в России (СССР), и на английском, если за пределами России (СССР), например РЕФАЛ и Java;
- для большинства понятий на русском языке при их первом появлении будет приведен англоязычный аналог.

Благодарности

Благодарность — это тяжкое бремя, лежащее
на нашей несовершенной природе.

Ф. Честерфилд

Хочу исполнить приятный долг и выразить благодарность всем тем, кто помог в работе над данной книгой. В первую очередь, это студенты математико-механического факультета СПбГУ, которые являлись терпеливыми слушателями различных курсов моих лекций и улучшали их своими вопросами и замечаниями.

Плодотворные длительные дискуссии с Денисом Черноивановым, Романом Шапошником, Михаилом Войтенко, Алексеем Духняковым и Дмитрием Петровым положительно повлияли на содержание книги. Дмитрий Петров также оказал значительную помощь в совершенствовании главы 2.

Глубокую признательность хочу выразить Игорю Некрестьянову за многочисленные обсуждения данной книги, настойчивые критические замечания и предложения, существенную поддержку и дружеское участие. Я благодарен своим коллегам — Юрию Юдину, Валерию Ушакову, Олегу Суходольскому, Екатерине Павловой и Алине Холчевой — за то, что они прочли черновые варианты этой книги. Их конструктивные замечания помогли улучшить изложение глав.

Я хотел бы поблагодарить руководителя проекта издательства "БХВ-Петербург" Евгения Рыбакова, который первый ознакомился с рукописью. Без его поддержки и настойчивости рукопись вряд ли оформилась бы в данную книгу. Хочу выразить благодарность Михаилу Кокореву за работу по редактированию книги.

Я буду благодарен и будущим критикам этой книги, которые, несомненно, появятся. Работа над текстом книги продолжается и она еще далека от завершения.

Игорь Одинцов

e-mail: **oio@nwgsm.ru**

Февраль 2002 г.

Предисловие ко 2-му изданию

Основные изменения в книге и в мире

Мафусаил жил 969 лет. Вы, дорогие мальчики и девочки,
в следующие десять лет увидите больше,
чем видел Мафусаил за всю свою жизнь.
Марк Твен

Прошло около двух лет с момента выхода первого издания. С определенной гордостью за книгу замечу, что подавляющая часть материала и на сегодняшний день остается актуальной. Тем не менее, мы постарались включить в книгу всю новую интересную информацию, которая появилась за это время. Значительная часть глав 1, 2, 3 и 8 переработана, добавлены новые разделы в главы 4—7. Исправлены замеченные в первом издании неточности и опечатки. Создан полноценный предметный указатель. Появились новые рисунки, ссылки на наиболее полезные книги, вышедшие за последние два года, а также новые ссылки на ресурсы сети Интернет. Учтены многие замечания и предложения, высказанные автору читателями книги (на электронный почтовый адрес пришло несколько сотен писем).

Первое издание книги задумывалось как энциклопедический справочник, свод по большей части беспристрастной информации, из которой каждый мог отобрать себе необходимую. Однако, по многочисленным просьбам читателей (особенно студентов), в данном издании появилось большое количество субъективных оценок автора, как непосредственно в тексте разделов, так и в итоговых разделах "Выводы и рекомендации" глав 2—7. Можно сказать, что теперь у читателя в руках не только "удочка", с помощью которой он будет ловить "рыбу", но и некоторое количество "рыбки" про запас на тот период, пока он не научится ее ловить полностью самостоятельно.

Изменения в компьютерном мире заслуживают отдельного анализа. Жестче стала "игра" компьютерных компаний, навязывающих покупателю свою продукцию, подчас сырую и некачественную. После разработки семиуровневой модели взаимодействия открытых систем (*см. разд. 7.2.3.1*) ряд аналитиков иронически замечал, что к уже существующим семи уровням можно добавить еще два: финансовый и политический. Сейчас очень многие платформы имеют эти два уровня в виде надстройки. Реальность и жесткость (жестокость?)

жизни такова, что по финансовым и политическим соображениям закрываются проекты, безотносительно к их реальным ценностям и пользе для программистского сообщества. Любой специалист в области разработки программного обеспечения должен быть готов к тому, что его проект будет закрыт по прихоти руководства компании, перебрасывающего силы на разработку новомодной идеи. Крупные компании давят на рынок, активно формируя нужный им спрос. Еще одно любимое занятие компаний по разработке программного обеспечения — постоянный выпуск новых версий, появление новых платформ и сред. Последнее время это сильно напоминает ситуацию в мире шоу-бизнеса, а не серьезную профессиональную работу.

Возникает вопрос: справедливо ли утверждение, что ничего нового за последние, например, 10 лет в области разработки программного обеспечения не появилось? В общем-то, нет. Можно найти достаточно много областей, в которых найдены методы и алгоритмы, позволившие сделать существенный прорыв. Типичный пример — новые алгоритмы, применяемые в поисковых системах в Интернете. Их использование значительно повысило адекватность (релевантность) запросу находимой в сети информации.

Впрочем, автор должен (фактически "по определению") являться оптимистом, как и большинство программистов. Следует помнить, что книга, утверждающая, что "во Вселенной есть жизнь", притягивает читателей гораздо сильнее, чем та, в которой более аргументировано говорится, что "во Вселенной нет жизни". Даже если обе книги написаны одним и тем же автором.

Книга как отправная точка

Мастеръ Гамбсъ этимъ полукресломъ
начинаетъ новую партію мебели.
1865 г. Санктъ-Петербургъ.
Илья Ильф и Евгений Петров

Очевидное, на наш взгляд, утверждение следует напомнить здесь. Профессиональным программистом можно стать после упорной работы и практики, а не после прочтения книги (пусть даже самой лучшей) или после взмаха волшебной палочки. Книга *поможет* Вам стать профессионалом, но не сделает Вас им автоматически. Начинающим программистам, стремящимся как можно скорее освоить программирование на некотором языке, можно предложить следующий план действий из трех пунктов.

- Прочитайте эту книгу.
- Прочитайте хорошую книгу с описанием языка программирования.
- Напишите хотя бы три достаточно серьезные программы на этом языке.

В последнем пункте можно использовать идею "подражания великим" и взять в качестве примеров аналоги существующих хорошо известных программ. Первая программа — файловый менеджер, аналог программы Norton

Commander (FAR manager). Заметим, что эта программа особенно любима многими отечественными программистами. Существует даже шуточное утверждение, что если за рубежом вы встречаете программиста и замечаете на экране его монитора хорошо знакомые синие панельки с желтыми буквами, то можете смело подходить и начинать разговор с ним по-русски. Впрочем, в каждой шутке есть доля шутки. Вторая программа — ftp-сервер, реализуя который Вы постигнете ряд тонкостей системного и сетевого программирования. Третьей задачей может стать реализация синтаксического анализатора (методом рекурсивного спуска) простого и красивого языка программирования Pascal. После ее написания у Вас появится понимание механизмов представления данных и работы с ними, реализованных в системах программирования.

Многие вопросы не удалось осветить в рамках этой книги. Автором готовится к изданию следующая книга, посвященная вопросу эволюции программных продуктов. В ней будет рассмотрено несколько подробных примеров разработки программных продуктов.

Благодарности

В первую очередь благодарю Вас, уважаемый читатель этой книги. Собственно, книга без Вас была бы также бессмысленна, как и разработанный программный продукт, не нашедший своего пользователя. Если Вы читаете ее впервые, надеюсь, она оправдает Ваши ожидания, а если Вы читали первое издание — спасибо за постоянство и доверие. Особая благодарность тем читателям первого издания, которые прислали мне письма. Я ответил всем.

Выражаю благодарность человеку, являющемуся легендой российского программирования, — члену-корреспонденту АН Святославу Сергеевичу Лаврову. Его "плотность критических замечаний" по рукописи книги была максимальной среди всех читателей и рецензентов. И если второе издание книги стало лучше, то во многом благодаря Святославу Сергеевичу.

Я благодарен профессору Санкт-Петербургского государственного университета Иосифу Владимировичу Романовскому, внимательно прочитавшему книгу и подготовившему обстоятельную рецензию.

Благодарю Александра Лукичева за многочисленные обсуждения книги. Его бескомпромиссность и стремление следовать только научному подходу вызывают глубокое уважение автора.

Игорю Некрестьянову, Александру Горшеневу, Денису Черноиванову и Алине Холчевой большое спасибо за прочтение черновых вариантов глав второго издания книги и высказанные замечания и комментарии.

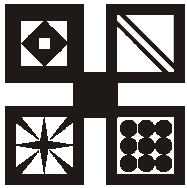
Алексея Лопатина, дважды чемпиона мира по программированию в составе команды Санкт-Петербургского государственного университета, благодарю за помощь в подготовке материала для раздела "олимпиадное программирование".

Студентам пятого курса кафедр информатики и системного программирования выпуска 2004 года спасибо за конструктивный критический анализ первого издания.

Всех сотрудников филиала "Эльбрус МЦСТ. Санкт-Петербург" и лично его директора Игоря Калошина благодарю за прекрасную творческую атмосферу, которая помогала созданию этой книги.

Игорь Одинцов

Февраль 2004 г.



Глава 1

Профессиональное программирование

В этой главе рассматриваются вопросы, над которыми рано или поздно задумываются все программисты.

- Каковы функции, черты и особенности мышления профессиональных программистов?
- Является программирование искусством, наукой, ремеслом или технологией?
- Что должны знать и уметь профессиональные программисты?
- Существуют ли стандарты для обучения программированию?
- Почему многие программные проекты выполняются с отставанием от графика, с превышением сметы расходов, при этом качество продукта не устраивает пользователя?

Также здесь приведены основные понятия и определения из области профессионального программирования.

1.1. Профессионализм в программировании

> Каждое очередное пятилетие приносит нам новый взгляд на программирование. Экстравагантный промежуточный этап в решении задачи, математическая головоломка, дело необычайной трудности, доступное лишь посвященным, своеобразное инженерное конструирование, особого рода логическое рассуждение, наконец, основа любой целеустремленной деятельности, вторая грамотность современного образованного человека.
>
> *А. П. Ершов*

Кто такой *профессионал*? Этим словом мы называем человека, который занимается каким-либо делом не просто как специалист, но и демонстрирует *профессионализм* — отличное владение своей профессией (в нашем случае —

программированием). Профессионализм — это интегральная личностная характеристика человека (программиста), который:

- овладел нормами профессиональной деятельности и общения и осуществляет их на высоком уровне, добиваясь профессионального мастерства в своей области (программировании);
- следует профессиональной ценностной ориентации, соблюдая профессиональную этику;
- развивает свою личность средствами профессии;
- стремится внести творческий вклад в профессию, обогащая ее опыт;
- стремится и умеет вызвать интерес общества к результатам своей профессиональной деятельности, способствует повышению веса и престижа своей профессии в обществе.

Образно можно заметить, что профессионализм делится на профессионализм деятельности и профессионализм личности. Профессионализм закладывается в процессе общего и профессионального образования, в отличие от мастерства, которое может быть накоплено в опыте и подражании. Природа профессионализма такова, что чем он выше, тем он у́же.

Наряду с понятием "профессионализм" используют также понятия "образованность" и "компетентность". Образованность и компетентность являются неотъемлемыми составляющими профессионализма.

Образованность — это наличие разносторонних знаний. В отличие от профессионализма, образованность чем выше, тем шире. Высокообразованный программист — это не только безукоризненный специалист в определенной области программирования, но и человек, имеющий широкий взгляд на современное программирование независимо от языков и платформ, которые он сейчас использует. Это также человек, уверенно ориентирующийся в других сферах науки и искусства, знающий философию, историю, владеющий иностранными языками.

Компетентность — это осведомленность в целом ряде компонентов, имеющих надпрофессиональный или внепрофессиональный характер и в то же время абсолютно необходимых сегодня в той или иной степени каждому программисту. Компетентность проявляется в таких качествах личности, как самостоятельность, способность принимать ответственные решения, творческий подход к любому делу и умение довести его до конца, коммуникабельность, умение вести диалог. Просто незаменимыми качествами программиста являются умение постоянно учиться и обновлять свои знания, гибкость мышления и наличие абстрактного, системного и экспериментального мышления.

Результаты работы программистов играют важнейшую роль во многих областях человеческой деятельности. Программирование привлекает как

престижностью этой профессии, так и потенциально высокими заработками. Некоторые люди, написав небольшие программы и не встретив при этом серьезных трудностей, решают, что программирование вообще не очень сложное занятие. Итак, нужны ли программисты-профессионалы в настоящее время, когда каждый выпускник школы может написать программу?

А нужны ли профессиональные писатели, когда все сами грамотные? Ответ очевиден. И мы надеемся, что эта книга поможет вам стать *настоящим* образованным и компетентным профессионалом.

1.1.1. Функции, черты и особенности мышления профессиональных программистов

1.1.1.1. Функции профессиональных программистов

Математик делает то, что можно, так, как нужно.
А программист — то, что нужно, так, как можно.
Программистский фольклор

Выделим основные функции профессиональных программистов [Цейтин 1990] [Буч, Рамбо, Джекобсон 2000].

- *Разработка систем большого объема*. Профессионалы знают, что "выход за некоторую границу сложности создаваемого программного продукта без надлежащей технологии разработки вообще невозможен". Системы большого объема создают новые принципиальные трудности. Интересно сравнение программирования больших систем со смоляными ямами доисторических времен, в которых увязали динозавры [Брукс 1999]: "Казалось, ничто в отдельности не вызывает трудностей — одну лапу всегда можно вытащить. Однако накопление действующих одновременно и взаимовлияющих факторов все более и более замедляет движение". Еще один образный пример приводится в книге по языку UML [Буч, Рамбо, Джекобсон 2000]: "Если вы хотите соорудить собачью конуру, то можете приступить к работе, имея в наличии лишь кучу досок, горсть гвоздей, молоток, плоскогубцы и рулетку... Хотя это и кажется комичным, многие компании, разрабатывающие программное обеспечение, хотят создать небоскреб, в то время как их подход к делу очень напоминает строительство собачьей конуры".
- *Создание системного и инструментального программного обеспечения*, которым будут пользоваться остальные. Профессиональными программистами создается инструментальное обеспечение в большом диапазоне — от профессионального до массового. Важная подзадача в этой области — создание языков, систем, интерфейсов, позволяющих пользователю работать в его собственных понятиях, а не в тех, которые навязаны

особенностями традиционного языка программирования или архитектурой компьютера.

- *Разработка программного продукта с предъявлением особых требований к его качеству и эффективности.* Различные требования предъявляются к программам, создаваемым только для себя, и к программам, предназначенным для большого количества пользователей. Во втором случае любая неаккуратность и малейшая небрежность может повториться многократно в работах будущих пользователей.
- *Разработка программ на основе подхода открытых систем. Открытая система* (open system) — система, которая состоит из компонентов, взаимодействующих друг с другом через стандартные интерфейсы. Общие свойства открытых систем таковы:
  - *мобильность* (переносимость) кода. Мобильность дает возможность легко переносить программное обеспечение на различные архитектурные и операционные платформы;
  - *интероперабельность.* Дает возможность легко осуществить обмен данными и взаимодействие с прикладными задачами в других системах;
  - *соответствие стандартам.* После того как стандарт принят, он становится силой, помогающей двигаться вперед. Как это ни парадоксально, но следование канонам дает новые степени свободы в современном мире.
- *Применение системного анализа*, включающего умение оценивать конкретные обстоятельства и следовать каждый раз наиболее приемлемому стилю. В голове программиста должны накопиться блоки знаний, чтобы он мог начать сравнивать эти блоки и мыслить блоками. Чтобы отличить структурное программирование от объектно-ориентированного программирования и оценить их пригодность для решения конкретных задач, надо иметь в голове и ту, и другую методологию.

1.1.1.2. Черты и особенности мышления программистов

Профессиональные программисты имеют свои черты, качества, способности и особенности мышления. Мы попытались сгруппировать наиболее важные из них, основываясь на работах [Цейтин 1990] [Дейкстра 1975] [Уэзерелл 1982] и собственном профессиональном опыте.

Первая группа черт и особенностей связана со способностью к абстрагированию и пониманию отношений между элементами.

- *Способность определить архитектуру программы*, т. е. разбить сложную задачу на элементарные составляющие и задать их комбинирование. Это основное качество программиста, связанное непосредственно с написанием программ.

- *Умение видеть задачу одновременно на разных уровнях детализации.* Программист должен свободно переходить от описания задачи в крупных понятиях к стоящим за этими понятиями сущностям более низкого уровня.
- *Умение представлять себе проектируемый процесс в динамике.* Обрабатываемые данные в некоторый момент времени могут иметь одни значения и взаимоотношения, а в следующий момент некоторые из них могут измениться.
- *Умение обобщать типичные ситуации.* Эта особенность означает, что необходимо уметь находить в программе идейно однородные участки. Для таких участков необходимо принять решение по их обобщению, выяснить его границы и выбрать наилучший способ программной реализации найденного решения. Это может быть как разработка функции или пакета, так и копирование в новую программу фрагментов из старой.
- *Умение применять и комбинировать хорошо известные программистские приемы и типовые алгоритмы.* Большинство новых идей должно находиться в тесном взаимодействии с уже известными идеями и методами. Простые и изящные решения типичных задач, возникающих в объектно-ориентированном программировании, приведены в книге [Гамма, Хелм, Джонсон, Влиссидес 2001].
- *Владение интеллектуальными средствами*, которые применяются для понимания программы. Эдсгер Дейкстра (Edsger Dijkstra) выделяет следующие три основных средства, которыми должен обладать профессиональный программист [Дейкстра 1975]:
  - *абстракция.* Абстракция проникла во все аспекты программирования. Алгоритм — это то, что остается от конкретных значений обрабатываемых данных. Переменная — абстракция соответствующего текущего значения. Абстракция присутствует в процессе присваивания операциям имен и использования этих операций. Язык программирования — это лишь средство описания абстрактных конструкций. Программист должен иметь способность полностью абстрагироваться от несущественных деталей, думая на нескольких уровнях абстракции одновременно;
  - *перечисление.* К перечислению прибегают, когда пытаются проверить правильность вычислений, которые сводятся к обозримому множеству последовательно выполняемых операторов. Сюда также включаются условные операторы выбора одного из двух или более вариантов;
  - *математическая индукция.* Это единственный способ рассуждений, который в конечном итоге позволяет работать с циклами и с рекурсивными процедурами.

Вторая группа черт и особенностей связана с гибкостью мышления, критичностью и самокритичностью. Сюда же мы отнесем готовность пополнять знания и переучиваться. В нее входят:

- *умение тщательно отбирать наиболее главное из всего имеющегося под рукой или приходящего в голову*. Эта черта, пожалуй, определяет основное отличие профессионализма от образованности;
- *наличие комплексного мышления*, включающего умение заранее определить этапы, которые нужно пройти, чтобы решить тот или иной вопрос;
- *способность анализировать собственные ошибки*. Эта черта указывает на требовательность программиста к себе, с одной стороны, и необходимость применения стиля программирования, уменьшающего количество ошибок, с другой;
- *способность при неудаче подавить самолюбие и поискать другой подход*;
- *способность выявлять все случаи, где можно применить теорию*, самостоятельно решиться на ее применение или обратиться за советом к более опытному программисту;
- *следование вопросам общечеловеческой этики*. Эти вопросы исследованы в книгах Достоевского, в том числе и в [Достоевский 1985]. Анализ одной из программистских этических проблем представлен в работе [Leveson-Turner 1993].

Третья группа черт и особенностей связана со склонностью к планированию, анализу и систематической работе.

- *Умение видеть дальше одной разрабатываемой в данный момент программы*. Следует учитывать широкое окружение данной частной задачи и возможность ее включения в некоторую общую систему. Считается, что эта черта в значительной степени отличает профессионального программиста от хакера, ориентированного на скорейшее получение конечного результата.
- *Культура собственного труда*, умение обеспечить себя необходимым инструментарием для работы.
- *Умение работать в коллективе*. Практически любая крупная разработка носит коллективный характер. Успех всей работы зависит от взаимопонимания, распределения функций и взаимоотношений в коллективе. Важно сочетать умение работать в коллективе с тем, что подсказывает индивидуальный опыт. Иногда программисту приходится передавать свой опыт другим членам коллектива, преодолевая их инерцию и давление руководства.
- *Умение работать с пользователем*. Профессиональный программист должен понимать потребности пользователя, уметь оценить удобство конкретных форм интерфейса, иметь возможность приучить пользователя к

новым средствам и системам. Программист должен обладать хотя бы минимумом психологических знаний, помогающих ему в общении с пользователем.

- Способность четко видеть действительные трудности и отбрасывать все, не относящееся к делу.

Перечисленные черты и особенности потребуются программисту в разных соотношениях на различных этапах разработки программы. Заметим, что программисты, в большей степени обладающие чертами первой группы, будут незаменимы в качестве архитекторов программных проектов.

1.2. Программирование. Что это такое?

Программирование — это то, чем занимаются программисты.
Программистский фольклор

"Программирование" — как и "любовь" — одно слово,
за которым скрывается бесконечное множество занятий.
Д. Вейнберг

Любовь — это все. И это все, что мы о ней знаем.
Э. Дикинсон

Для того чтобы определить предмет наших рассуждений, укажем то место, которое он занимает в такой предметной области, как информатика (рис. 1.1).

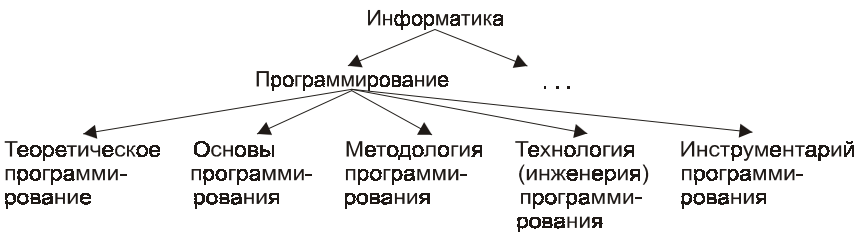


Рис. 1.1. Предметная область информатики

Информатика — наука, изучающая законы и методы накопления, передачи и обработки информации с помощью компьютера.

Программирование — раздел информатики, изучающий описание процессов обработки данных. В программировании четко выделяются следующие подразделы.

- *Теоретическое программирование*, объектом изучения которого являются математические абстракции программ. Начало этому направлению положили работы отечественных ученых, относящиеся к 1950-м годам. Инициатором работ был Алексей Андреевич Ляпунов, увидев-

ший в программировании самостоятельную область знания, требующую создания собственной теории. Он предложил *операторный метод* программирования, в котором впервые была предпринята формализация понятия программы и поставлена проблема разработки эквивалентных преобразований программ. В качестве объектов, на которых можно строить эквивалентные преобразования, были введены *схемы программ*, сохраняющие управляющую структуру программ и отвлекающиеся от детального описания операторов и логических условий. Отдельным направлением в теории схем программ стала теория алгебраических моделей программ.

О теоретическом программировании

Теоретическое программирование часто называют "фундаментальной информатикой". С одной стороны, это та часть информатики, в которой приложения сегодняшнего дня жертвуются в обмен на вероятность того, что когда некоторая теория понадобится, она уже будет развита в достаточной степени. С другой стороны, потребителями эквивалентных преобразований программ, разработанных на алгебраических схемах программ, стали такие области, как оптимизация и верификация программ, а также их защита и маскировка.

- *Основы программирования*, включающие концепции и навыки, которые важны для практики программирования в качестве фундаментальных основ:
  - структуры данных, их представление в памяти компьютера;
  - информационный поиск, упорядочивание линейных массивов и файлов;
  - формальные языки, грамматики, автоматы и другие абстрактные машины;
  - синтаксический анализ программ;
  - оценку трудоемкости и теорию сложности алгоритмов;
  - спецификацию задач, доказательство свойств программ, автоматический синтез программ;
  - семантику языков программирования.
- *Методология программирования*, изучающая методы с точки зрения основ построения. Это объединенная единым философским подходом совокупность методов, применяемых в процессе разработки программного обеспечения.

О том, что такое метод

Метод — путь исследования или познания, теория, учение. Метод включает средства (с помощью чего осуществляется действие) и способы (каким образом осуществляется действие).

- *Технология программирования*, изучающая технологические процессы и порядок их прохождения (с использованием знаний, методов и средств).

Технологический процесс — последовательность направленных на создание заданного объекта действий (технологических операций), каждое из которых основано на каких-либо естественных процессах и человеческой деятельности. Обратим внимание на то, что знания, методы и средства могут использоваться в разных процессах и, следовательно, технологиях.

Об использовании понятия "технология"

Очень часто мы слышим такие сочетания, как "CASE-технология", "Интернет-технология", "Java-технология" и т. п. Они не очень корректны и применяются, как правило, в рекламных и маркетинговых акциях. Будем считать, что в таких сочетаниях слово, употребляемое рядом со словом "технология", подчеркивает специфику определенного средства, поддерживающего ведение технологических процессов, например набора инструментов, совокупности стандартов или языка программирования, и связанных с ним механизмов.

Новосибирский академик А. П. Ершов в конце 1960-х годов предложил переводить "Software Engineering" русскоязычным термином "технология программирования". Наряду с ним широко используется и термин "инженерия программирования", причем в большинстве случаев мы можем считать, что эти термины обозначают одно и то же. Если говорить об отличиях, то область инженерии программирования (как инженерного дела, творческой технической деятельности) более широкая и охватывает весь спектр практических проблем создания программного обеспечения. *Инженерия программирования* изучает различные методы и инструментальные средства с точки зрения определенных целей, т. е. имеет очевидную практическую направленность. Основная идея инженерии программирования в том, что разработка программного обеспечения является формальным процессом, который можно изучать и совершенствовать. Данный раздел имеет четко определенную дату рождения — 1968 год. Причина его появления — реакция на так называемый "кризис программного обеспечения". Характерные вопросы и задачи инженерии программирования, изложенные Фредериком Бруксом (Frederick Brooks) [Брукс 1999], актуальны и по сегодняшний день.

- Как проектировать и строить программы, образующие *системы*?
- Как проектировать и строить программы и системы, являющиеся надежным, отлаженным, документированным и сопровождаемым *продуктом*?
- Как осуществлять интеллектуальный контроль в условиях большой *сложности*?

Сегодняшнее содержание раздела "инженерия программирования" весьма динамично и отражает большое количество направлений исследования.

Приведем список некоторых из них, которые активно обсуждаются на международных конференциях по инженерии программирования:

- процесс разработки программного обеспечения;
- формирование требований к программному продукту;
- обратная (реверсивная) инженерия — процесс восстановления спецификации из кода;
- тестирование;
- сопровождение и эволюция программного обеспечения;
- архитектура программного обеспечения;
- объектно-ориентированное моделирование;
- инженерия программирования и "промежуточное" (между сетевой операционной системой и программой пользователя) программное обеспечение;
- анализ программного обеспечения;
- формальные спецификации;
- математические основания инженерии программирования;
- инженерия программирования и производительность программного продукта;
- инженерия программирования для систем реального времени;
- инженерия программирования для мобильных систем;
- инструментарий и окружение инженерии программирования;
- инженерия программирования и Интернет;
- экономика программного обеспечения;
- метрики программного обеспечения;
- инженерия программирования как учебная дисциплина.

❐ *Инструментальное направление*, изучающее системы программирования. Сюда входят все инструменты, поддерживающие процесс программирования.

1.3. Программирование. Чему учиться?

Двадцать лет Ли Чанг учился убивать страшных драконов. И когда он овладел этим искусством в совершенстве, оказалось, что драконов на свете нет. И тогда он стал учить этому других.

Китайская мудрость

В предыдущем разделе мы в общих чертах определили, что представляет собой программирование. У многих может возникнуть вопрос: чему же конкретно надо учиться, чтобы стать профессионалом?

Мы рассмотрим четыре подхода, связанных с обучением.

- Точка зрения на программирование международных организаций, занимающихся стандартизацией.
- Точка зрения российских программ по информатике.
- Точка зрения программистов с большим стажем работы.
- Подход, предлагаемый данной книгой.

1.3.1. Подход на основе рекомендаций ACM/IEEE

Начнем мы с изучения точки зрения международных организаций, занимающихся стандартизацией:

- Ассоциации по вычислительной технике (The Association for Computing Machinery — ACM, **http://info.acm.org**). Это международная научная и образовательная организация, основная задача которой — распространение навыков, теорий и приложений из области информационных технологий;
- Института инженеров по электротехнике и электронике (Institute of Electrical and Electronic Engineers — IEEE, **http://www.ieee.org**). Это всемирная организация инженеров, направляющая деятельность технических обществ и участвующая в разработке стандартов.

Результатом совместной работы этих организаций стали "Рекомендации по преподаванию информатики 2001" (**http://www.computer.org/education/cc2001**). Рекомендации исправляют и дополняют более ранние отчеты, в первую очередь наиболее известные рекомендации 1991 г. [Curriculum 1968], [Tucker 1991], [Denning 1989], [Parnas 1990]. Структура совокупности знаний по информатике организуется в виде трехуровневой иерархической структуры: областей (отдельных частей дисциплины информатики), разделов (тематических модулей внутри области) и набора тем для каждого раздела.

Специальная комиссия определила 14 областей, структурировав тем самым всю предметную область информатики. В табл. 1.1 приведен перечень этих областей с указанием обязательных разделов. Обязательные разделы включают в себя только тот материал, по которому достигнуто практически всеобщее согласие о необходимости данного раздела каждому студенту в каждой программе обучения компьютерной специальности.

Обязательные разделы дополняются факультативными. Например, для области "алгоритмы и теория сложности", одним из факультативных разделов является "теория автоматов". Отсутствие данного важного раздела среди обязательных вовсе не означает негативного мнения о его ценности. Просто не было достигнуто единодушие в рабочей группе, которое позволило бы отнести раздел к категории обязательных.

Таблица 1.1. *Перечень рекомендованных ACM/IEEE модулей знаний*

Область	Разделы
Дискретные структуры (DS)	Функции, отношения и множества. Основы логики. Методы доказательства. Основы вычислений. Графы и деревья. Дискретная вероятность
Основы программирования (PF)	Основные конструкции программирования. Алгоритмы и решение задач. Фундаментальные структуры данных. Рекурсия. Событийно-управляемое программирование
Алгоритмы и теория сложности (AL)	Основы анализа алгоритмов. Алгоритмические стратегии. Фундаментальные вычислительные алгоритмы. Распределенные алгоритмы. Основы теории вычислимости
Архитектура и организация ЭВМ (AR)	Цифровая логика и цифровые системы. Представление данных в памяти компьютера. Организация машины на уровне ассемблера. Устройство памяти компьютера. Взаимодействие и коммуникации. Функциональная организация. Многопроцессорные и альтернативные архитектуры
Операционные системы (OS)	Обзор операционных систем. Основы операционных систем. Параллелизм. Планирование и диспетчеризация. Управление памятью. Управление устройствами
Распределенные вычисления (NC)	Введение в распределенные вычисления. Сети и телекоммуникации. Сетевая безопасность. Web как пример архитектуры "клиент-сервер"
Языки программирования (PL)	Обзор языков программирования. Виртуальные машины. Введение в трансляцию. Переменные и типы данных. Механизмы абстракции. Объектно-ориентированное программирование
Взаимодействие человека и машины (HC)	Основы взаимодействия человека и машины. Построение простого графического интерфейса
Компьютерная графика и визуализация (GV)	Фундаментальные методы в графике. Графические системы
Интеллектуальные системы (IS)	Основные вопросы, связанные с интеллектуальными системами. Поиск решений. Представление знаний и вывод
Управление информацией (IM)	Информационные модели и системы. Системы баз данных. Моделирование данных
Социальные и профессиональные вопросы (SP)	История информатики. Социальный контекст информатики. Методы и средства анализа. Профессиональная и этическая ответственность. Недостатки компьютерных систем и риски, связанные с их применением. Интеллектуальная собственность. Конфиденциальность и гражданские свободы

Таблица 1.1 *(окончание)*

Область	Разделы
Инженерия программирования (SE)	Проектирование программного обеспечения. Использование программных интерфейсов приложений. Программные средства и окружения. Процессы разработки программного обеспечения. Спецификация и требования. Проверка соответствия. Эволюция программного обеспечения. Управление программными проектами
Вычислительная математика и численные методы (CN)	–

Поскольку для каждого из обязательных разделов указывается рекомендуемое время, можно определить 5 областей-лидеров: дискретные структуры, основы программирования, алгоритмы и теория сложности, архитектура и организация ЭВМ, а также инженерия программирования. Тем самым, безоговорочным лидером становятся основы программирования, подкрепленные дискретной математикой с одной стороны, архитектурной платформой с другой и практической направленностью разработки программных продуктов с третьей.

В качестве основы для разработки учебной программы по информатике рекомендации определяют набор общих требований, предметный материал, обязательный для создания полного представления о специальности. Они не навязывают той или иной структуры курсов. Общие требования представлены в виде модулей знаний. При составлении конкретных учебных программ, помимо обязательного включения материала модулей, должны учитываться следующие составные части:

- изучение математики и естественно-научных дисциплин;
- проведение лабораторных работ, включая программирование;
- воспитание профессиональных навыков.

1.3.2. Российские программы по информатике

Старайся прежде быть мудрым, а ученым —
когда будешь иметь свободное время.

Пифагор

Российские программы по информатике в большой степени соответствуют рекомендациям ACM/IEEE. Одной из наиболее известных и удачных отечественных программ является учебный план по специальности 220400 "Программное обеспечение вычислительной техники и автоматизированных

систем" [ГОС 220400] [Лавров, Слисенко, Цейтин 1985]. Сейчас набирает популярность среди студентов и преподавателей новая специальность 351500 "Математическое обеспечение и администрирование информационных систем". В последнее время российскими специалистами был организован ряд конференций, посвященных преподаванию информатики, например открытая всероссийская конференция "Преподавание информационных технологий в Российской Федерации" (**http://www.it-education.ru/**).

Основные направления развития информатики в целом соответствуют тем специальностям, которые выделены в российских стандартах. Российские стандарты и программы университетов многих стран мира в целом схожи друг с другом, что отражает тот факт, что видение тенденций развития информатики в России и в других странах практически идентично.

Однако существуют и некоторые особенности российского образования в области информатики. В нем, по сравнению с американской системой образования, придается большее значение фундаментальному математическому образованию и несколько меньшее значение естественным наукам и профессиональной подготовке. Это связано с тем, что компьютерные науки в России создавались преимущественно математиками, традиционно отдающими предпочтение фундаментальным знаниям. Овладение такими знаниями в наибольшей степени способствует формированию мышления обучающихся. Причина различий здесь в том, что в России компьютерные науки и соответствующие образовательные программы исторически преимущественно математические, а в Америке — инженерные.

Следствием этого является то, что американский стиль обучения ориентирован на изучение процесса набора стандартных решений под типовые ситуации. Фактически, это подготовка технологов, обеспечивающих работу серийного производства. Российский стиль обучения подразумевает осмысление принципов получения решения и понимание важности развития решения задачи до обобщенного подхода. Это подготовка ученых, способных решать уникальные задачи.

Конечно, большой объем часов, отводимый на общематематические дисциплины, имеет и отрицательные стороны. Главная из них — уменьшение для студента возможности выбора. Обучение должно быть вариативным — с множеством альтернативных спецкурсов при небольшом числе обязательных.

Отметим также, что большая часть того, чему Вас будут учить (где бы то ни было), в будущей работе Вам не понадобится, как бы тщательно ни был составлен учебный план. В хороших учебных заведениях студент начинает это понимать, придя на производственную практику в компанию, занимающуюся разработкой программного обеспечения. И наоборот, Вам могут потребоваться в лучшем случае месяцы, чтобы научиться тому, что требуется для успешной работы. Поэтому главное, что требуется от учебного заведения — это *научить учиться*.

1.3.3. Точка зрения опытных программистов

Мнения общественных организаций и специалистов в области обучения дополним мнением опытных программистов. Нам также будет интересно разобраться, насколько эти мнения близки друг другу.

Тимоти Летбридж (Timothy Lethbridge) в 1998 году провел опрос более 200 опытных программистов [Lethbridge 2000]. Было выбрано 75 предметных областей, по которым есть учебные курсы практически в каждом высшем учебном заведении. Относительно каждой области был задан ряд вопросов.

- Насколько полезным оказалось знание *подробностей* именно этого предметного материала в Вашей карьере разработчика программного обеспечения или менеджера программных проектов?
- Какое *влияние* оказало изучение этого предметного материала на Ваше мышление, независимо от непосредственного применения подробностей на практике?

Ответы давались в баллах в диапазоне от нуля (бесполезность или отсутствие влияния) до пяти (существенная польза или значительное влияние). Некоторые результаты опроса представлены в табл. 1.2. В графе "Полезность" вычислено среднее арифметическое значение граф "Подробности" и "Влияние". Предметные области перечислены в порядке убывания их значимости для опрашиваемых.

Таблица 1.2. *Наиболее полезные области программирования*

Предметная область	Полезность	Подробности	Влияние
Конкретные языки программирования	3.8	4.1	3.5
Структуры данных	3.6	3.7	3.4
Проектирование программ и паттерны	3.5	3.6	3.5
Архитектура программного обеспечения	3.4	3.5	3.3
Техническое задание и спецификация	3.4	3.5	3.3
Человеко-машинное взаимодействие	3.3	3.3	3.3
Объектно-ориентированный подход	3.3	3.3	3.3
Этика и профессионализм	3.3	3.2	3.4
Методы анализа и проектирования	3.3	3.3	3.3
Выступления и презентации	3.3	3.5	3.1
Управление проектами	3.3	3.4	3.2

Таблица 1.2 *(окончание)*

Предметная область	Полезность	Подробности	Влияние
Тестирование, верификация и качество	3.2	3.3	3.1
Проектирование алгоритмов	3.2	3.3	3.1
Написание технической документации	3.1	3.4	2.9
Операционные системы	3.1	3.3	3.0
Базы данных	3.1	3.3	2.8
Руководство проектами	3.0	3.1	3.0
Конфигурация и управление версиями	3.0	3.3	2.8
Передача данных и сети	3.0	3.1	2.8
Основы менеджмента	2.9	2.9	2.9

Результаты данного опроса не вызывают удивления. Программистов-практиков "кормит" знание конкретных языков программирования. Далее идут различные разделы основ программирования и технологии программирования.

Еще один опрос был проведен в 1999 году среди руководителей большого количества американских компьютерных компаний. Им было предложено оценить перспективность основных практических областей программирования в процентах. В этом опросе практические области включали названия конкретных языков и систем программирования, операционных систем и баз данных. Отметим, что на то время почти половина опрошенных назвала наиболее перспективной областью "Язык программирования C++". Далее, со значительным отрывом, в порядке убывания голосов шли: "Объектно-ориентированное проектирование", "Операционная система Windows", "Язык и система программирования Visual Basic" и "Проектирование графического пользовательского интерфейса". Добавим, что "Язык программирования Java" также вошел в десятку, но всего лишь десятым. И опять же, точка зрения руководителей в достаточной степени понятна и объяснима. Язык C++ был и остается языком профессионального промышленного программирования, несмотря на все его недостатки. Руководители в большей степени, чем программисты, прагматичны и трезво оценивают реальности рынка и факторы, снижающие риск (например, наличие стандарта языка). И шумиха, связанная с языком Java, их заражает в меньшей степени, чем программистов-оптимистов.

1.3.4. Подход, предлагаемый данной книгой

Нам представляется, что настоящий программист, специалист в области информатики должен обладать знаниями в трех следующих областях, уметь их гармонично сочетать и применять на практике:

- профессиональное программирование;
- теория программирования;
- математические основы программирования.

1.3.4.1. Профессиональное программирование

Данная книга целиком посвящена элементам профессионального программирования. В предисловии мы уже отметили, что основные области в профессиональном программировании представлены тремя взаимосвязанными группами. Каждая группа — это объединение двух достаточно близких направлений. Образно это можно представить в виде трех кофейных зерен (рис. 1.2). Все эти направления имеют четкую практическую направленность.

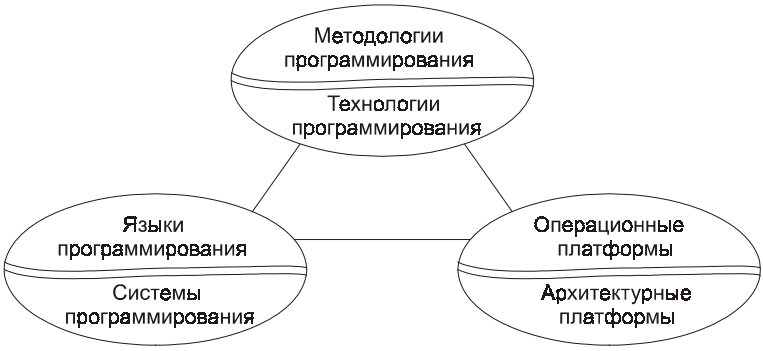


Рис. 1.2. Три зерна профессионального программирования

1.3.4.2. Теория и фундаментальные основы программирования

Кроме профессиональных знаний для программиста важны еще теоретические и фундаментальные знания. Это направления программирования, связанные с тремя основными понятиями: алгоритм, задача и вычислитель *(см. разд. 1.7.3)*. На рис. 1.3 представлены некоторые эти направления, причем они расположены между тремя основными понятиями так, чтобы показать их близость к одному или нескольким из них.

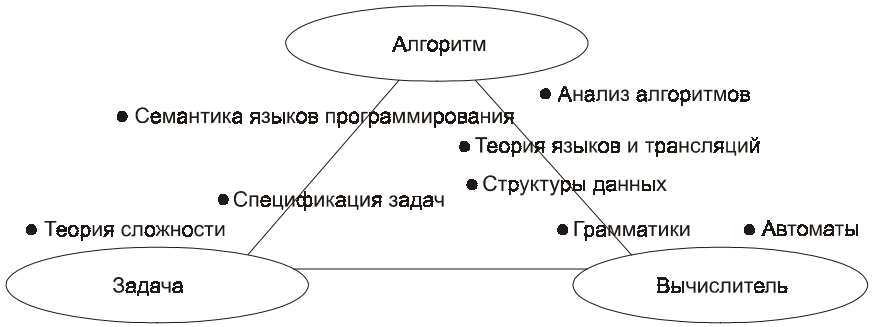


Рис. 1.3. Направления теории программирования

В качестве литературы по теоретическим и фундаментальным знаниям в области программирования предложим четыре монографии.

- "Искусство программирования" [Кнут 2000].
- "Алгоритмы: построение и анализ" [Кормен, Лейзерсон, Ривест 1999].
- "Алгебраическая алгоритмика" [Ноден, Китте 1999].
- "Введение в теоретическое программирование" [Ершов 1977].

1.3.4.3. Математические основы программирования

Когда Дональд Кнут (Donald Knuth) задумал серию книг под названием "Искусство программирования", то он отметил необходимость для программирования важных математических инструментов. Была необходима именно та математика, которая требуется для досконального и обоснованного истолкования компьютерных программ. Как правило, это математика, которая имеет конкретные классические результаты, позволяющая получить удовольствие от решения численных задач или оценить по достоинству роль математических методов. При этом важно уметь исследовать с математической точки зрения как дискретные, так и непрерывные (континуальные) системы. На рис. 1.4 представлены разделы математики, наиболее важные для программирования.

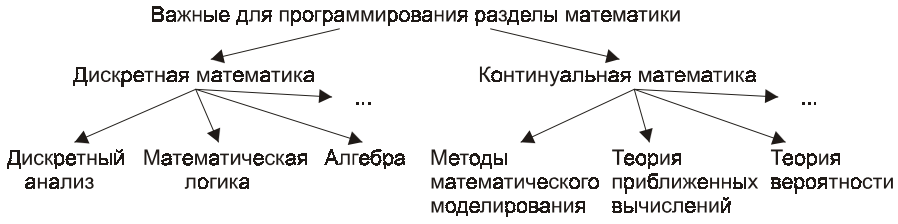


Рис. 1.4. Математические основы программирования

Следующие четыре учебника представляют достаточно полную картину в области математических основ программирования:

- "Конкретная математика. Основание информатики" [Грэхем, Кнут, Паташник 1998];
- "Дискретный анализ" [Романовский 1999];
- "Комбинаторика для программистов" [Липский 1988];
- "Программирование. Математические основы, средства, теория" [Лавров 2001].

Добавим, что программистам важно объяснять то, как математика находит применение в информатике. Например, чтобы оценить отказы вычислительной техники, используется теория вероятностей, а таких примеров очень много и в других разделах математики.

1.4. Программирование. Как учиться?

Если ты знаешь все, тебе нужно еще многому научиться.
Народная мудрость

Итак, мы увидели, чему надо учиться. Теперь перейдем к вопросу: как должно происходить обучение программированию. Далее мы изложим следующие подходы:

- первый основан на рекомендациях ACM/IEEE и представляет собой следование соответствующим мировым стандартам;
- второй представляет собой рекомендации российских специалистов;
- третий отражает авторскую точку зрения известных специалистов на формирование навыков программирования;
- наконец, мы обсудим роль наставника в процессе обучения программированию.

1.4.1. Подход на основе рекомендаций ACM/IEEE

Рассмотрим методологический подход на уровне следования рекомендациям ACM/IEEE. Наиболее интересными здесь представляются стратегии построения обучения программированию. Определяется шесть различных реализаций вводных курсов и четыре тематических подхода к компоновке основных курсов. На рис. 1.5 представлена взаимосвязь их в структуре учебного плана.

Шесть вводных курсов должны знакомить с набором фундаментальных концепций информатики.

- *Подход с ориентацией на императивное программирование* является наиболее традиционным.

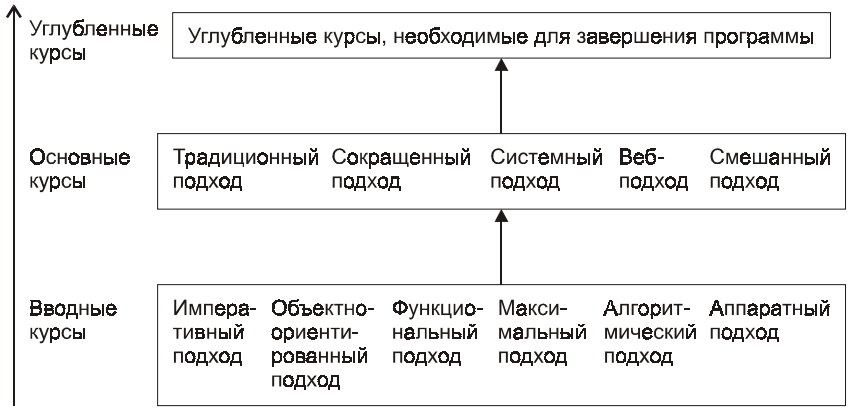


Рис. 1.5. Уровни курсов и стратегии обучения

- *Подход с ориентацией на объектно-ориентированное программирование* с самого начала делает акцент на принципах объектно-ориентированного проектирования и программирования. Первый курс начинается непосредственно с понятия объектов и наследования.
- *Подход с ориентацией на функциональное программирование* позволяет сконцентрироваться на фундаментальных вопросах, благодаря минималистскому синтаксису функциональных языков.
- *Подход с максимальным охватом материала* предоставляет более целостный взгляд на дисциплину. При таком подходе обучающиеся сразу могут оценить объем информатики.
- *Подход с ориентацией на алгоритмы* предполагает представление основных концепций информатики с использованием псевдокода вместо реального языка программирования. Данный подход минимизирует усилия, уходящие на изучение специфических синтаксических конструкций.
- *Подход с ориентацией на аппаратную платформу* учит основам информатики, начиная с машинного уровня и продвигаясь затем к более абстрактным концепциям. Считается, что таким образом студенты могут изучить информатику последовательно и без каких-либо мистификаций.

Преподавание основного уровня может быть построено на основе одного из четырех основных вариантов реализации и дополнительного смешанного:

- *традиционный подход*, в котором отдельные курсы посвящены самостоятельным темам;
- *сокращенный подход*, который организовывает курсы вокруг более общих тем. Материал связанных между собой областей информатики будет собран в тематические курсы, представляющие собой единое целое;

- *интенсивный системно-ориентированный подход.* В качестве объединяющей темы используется разработка систем. Такая модель включает в себя значительно больше технического и профессионального материала, чем другие модели;
- *подход с ориентацией на Веб*, использующий сеть в качестве основного лейтмотива. Эта модель выросла из массового спроса на курсы, которые больше внимания уделяют Интернету и Вебу;
- *смешанный подход,* объединяющий элементы двух или более подходов для создания нового смешанного плана, отвечающего интересам и требованиям конкретного учебного заведения.

Все указанные подходы в качестве общей цели представляют собой основные идеи и устоявшиеся концепции информатики, которые должен знать каждый для успешной работы в этой области.

Следующая ступень — *углубленные курсы*, содержание которых выходит за рамки обязательного материала. Здесь учебные заведения могут ориентироваться на свои области наилучшей компетенции, определяемые потребностями студентов, знаниями преподавателей, нуждами и запросами сообщества пользователей.

1.4.2. Рекомендации российских специалистов

Арифметика мозги в порядок приводит.
Старинный русский фольклор

Первая группа рекомендаций основывается на том факте, что в программировании присутствуют одновременно математические, естественно-научные и инженерные корни. Обучающиеся должны познакомиться с тремя различными подходами, используемыми в своей профессиональной деятельности исследователями и специалистами-практиками.

- Первый из этих подходов — *теория* — присущ математике и включает в себя следующие элементы:
  - определения и аксиомы;
  - теоремы;
  - доказательства;
  - интерпретацию результатов.

  Этот подход применяется в собственно математических дисциплинах и при изучении теоретических основ информатики.
- Второй подход — *абстракция* — присущ естественным (экспериментальным) наукам и включает в себя следующие элементы:
  - сбор данных и выработку гипотез;
  - моделирование и предсказание;

- планирование эксперимента;
- анализ результатов.

В программировании постоянно создаются модели, определяются структуры данных, предлагаются новые архитектуры и т. д. Формулируются гипотезы относительно этих моделей, проверяются альтернативные проектные решения и даже теоретические построения.

- Третий подход — *конструирование* — присущ инженерным дисциплинам, заключается в создании системы или устройства для решения поставленной задачи и включает в себя следующие элементы:
  - определение требований;
  - написание спецификаций;
  - разработку и реализацию;
  - тестирование и анализ.

  Обучающиеся должны не только создавать самостоятельные разработки (по возможности, реальных систем), но и изучать разработки других. От них нужно требовать оценки производительности и стоимости, а также изучения альтернативных решений.

Вторая группа рекомендаций советует обучающимся получать подготовку в следующих трех направлениях по каждому предмету:

- курс лекций;
- практические занятия с преподавателем;
- практикум (самостоятельная работа).

1.4.3. Авторские подходы

1.4.3.1. Подход Г. С. Цейтина

Методологический подход Григория Самуиловича Цейтина исходит из того, что целью курса программирования следует считать переход от непонимания программирования к его пониманию. Тогда для формирования навыков программирования нужно изучить их структуру у подготовленных программистов и определить содержание курса в соответствии с выделенными элементами. Предполагая, что программирование — это переход от знания о задаче, выраженного в обычной, не программной форме, к выражению этого знания (точнее, части его) в форме программы, предлагается выделить три элемента программирования по такому преобразованию знания.

- Первый элемент — *переход от знания в математической форме* к знанию в форме программы. Ему соответствует программирование математических формул, программирование рекуррентно решаемых задач посредством

рекурсивных процедур (во всех этих случаях не требуется даже присваивания) и элементы доказательного программирования, в частности, использование инвариантов цикла.

- Второй элемент — *моделирующий*, включает построение последовательности действий в программе по образцу последовательности действий в моделируемой реальной системе или моделируемом поведении человека, решающего задачу стандартным способом (именно здесь нужна память и присваивания). Примерами такого подхода могут быть задачи моделирования физических процессов. В рамках этого же подхода можно научиться использованию стандартных элементов программ, например циклов для суммирования.
- Третий элемент программирования — *преобразования программ*. Они включают сборку сложной программы из элементарных программ и эквивалентные преобразования. Конструкции используемого языка программирования излагаются не по логике описания этого языка, а в связи с их конкретным использованием (например, массив как представление вектора или матрицы в линейной алгебре и массив как способ хранения промежуточных результатов с целью избежания повторного вычисления рассматриваются порознь).

1.4.3.2. Подход М. Шоу

Подход, предложенный Мэри Шоу (Mary Shaw), направлен на улучшение стратегии построения курсов обучения программированию (**http://spoke.compose.cs.cmu.edu/shaweb/edpapers/crn.htm**). Альтернативные модели построения курсов пытаются сделать цели более ясными. Мэри Шоу предлагает организовывать курсы вокруг идей, а не вокруг артефактов (искусственно созданных фактов). Кстати, основными артефактными динозаврами программирования являются курсы "Операционные системы" и "Компиляторы". Это согласуется и с подходами, принятыми в других науках. Простейший пример: студенты изучают не структуру парового котла, а термодинамику.

Студенты, изучающие курс построения компиляторов, могут считать, что он посвящен только разработке компиляторов. На самом деле существует большое количество других целей, которые помогает решать этот курс. Перечислим некоторые из них:

- изучение построения структуры хорошо исследованных систем среднего размера;
- организация взаимодействия нескольких модулей;
- исследование некоторых неочевидных алгоритмов и структур данных (связанных с символьными таблицами, деревьями разбора и обходом графов);
- решение задачи применения на практике хорошо исследованных теоретических областей (например, таких как синтаксический анализ).

1.4.4. Отступление "О роли наставника в процессе обучения"

Эмпирически найденные правила хорошего программирования передаются от мастера к ученикам. Наставничество существовало и существует практически во всех специальностях, в том числе и в программировании.

Классическое обучение любому делу состояло в том, что ученик в течение многих лет выполнял вспомогательные операции, перенимая основные приемы у более опытных работников. Постепенно на него возлагались более серьезные обязанности, и после формальной проверки навыков он получал свидетельство о своей профессиональной компетентности. Бывший ученик становился ремесленником, сам получал заказы и выполнял работу. Рано или поздно он представлял на суд гильдии свои лучшие работы и мог подняться на высшую ступень профессиональной иерархии, становясь мастером.

Наиболее развита традиция наставничества в России и в Японии. Японцы в целом расположены к помощи друг другу. Они не привыкли держать свои методы и приемы работы в тайне, а предпочитают учиться друг у друга. Эта традиция называется "минарай" — наблюдение за опытными рабочими с целью освоения их навыков. Интересно то, что многие японцы не доверяют официальному обучению в учебных заведениях, поскольку оно построено по западным образцам. Они считают, что официальное обучение дает только основы специальности и развивает общие способности, но научиться тому, что им действительно нужно, можно только в процессе работы и через индивидуальное обучение.

Особой разновидностью наставничества может быть парное программирование, особенно известное в последнее время благодаря некоторым популярным технологическим подходам. В случае парного программирования в пару к опытному программисту добавляется программист-новичок. Самое ценное, что может сделать опытный разработчик, это передать свои знания программистам-новичкам. Новичок перенимает опыт, учится совместной работе и общению. При этом существенно увеличивается обмен информацией. И опытные программисты, и новички утверждают, что они при такой организации, кроме всего прочего, получают больше удовольствия от своей работы.

Конечно, далеко не все компании принимают идею наставничества. Во многих случаях новичков бросают в "свободное плавание". Кто "выплывет" — останется работать в компании, а тем, кому не удалось, придется искать новое место. Тем не менее, будем утверждать, что будущее за теми компаниями, которые бережно относятся к "человеческому фактору" и в дальнейшем получают существенную отдачу и благодарность от выращенных специалистов.

Завершая этот раздел, подчеркнем, что наиболее ценным качеством наставника является не столько способность преподнести ученику готовый результат, сколько познакомить его с *процессом поиска* этого результата.

1.5. Программирование. Искусство, наука, ремесло или технология?

Существует предмет жарких споров, в которые вовлечены как опытные программисты, так и новички. Чем является программирование в большей степени: искусством, наукой, ремеслом или технологией? Приведем мнения классиков программирования, каждый из которых выражает свое отношение к этой проблеме.

- Дональд Кнут в лекции "Программирование как искусство" в качестве лауреата премии Тьюринга [Кнут 1974] сделал интересный анализ соотношения искусства и программирования: "Программирование — это искусство, поскольку оно является приложением накопленных знаний для практических целей, поскольку оно требует умения и мастерства, и в особенности потому, что продукты программирования могут представлять эстетическую ценность. Программист, который бессознательно ощущает себя художником, получает удовольствие от своей работы и справляется с ней лучше".
- Дэвид Грис (David Gries) начинает свою книгу "Наука программирования" [Грис 1984] со следующего утверждения: "Программирование начиналось как искусство; даже сейчас большинство учится ему, наблюдая, как работают другие (например, преподаватель или более опытный коллега), постигая приемы и мало задумываясь над принципами, которые лежат в их основе. Однако в результате научных исследований последнего десятилетия найдены некоторые полезные теоретические положения и общие принципы, так что наступает время, когда можно начинать учить принципам и их осознанному применению. В этом учебнике я сделал попытку поделиться своим пониманием возникающей на наших глазах науки программирования и восхищением ею".
- В то же время первые слова книги Чарльза Уэзерелла (Charles Wetherell) "Этюды для программистов" [Уэзерелл 1982] звучат так: "Программирование — это ремесло, и каждый программист должен достичь нужного профессионального уровня".
- Игорь Васильевич Поттосин в статье, посвященной академику Ершову, пишет: "Ершов первым (во всяком случае, среди отечественных программистов) увидел другую ипостась программирования — не исследовательскую, а производственную — как основу новой возникающей промышленности, софтверной промышленности. Очень важно, что Ершов предвидел роль и значимость этой промышленности, необходимость

готовить ее кадры, причем разноплановые: архитекторов, руководителей проектов, тестовиков, разработчиков. Иногда он сознательно обострял свою позицию — говорил о "дегероизации трансляции" в том смысле, что создание "хороших" трансляторов с известных языков пусть и сложная, но конструкторская, а не исследовательская задача; ввел термин "фабрика трансляторов". Такая "промышленная" точка зрения вызвала резкие возражения среди многих, и Ершов достаточно долго и упорно пробивал свою позицию, которая, как оказалось впоследствии, была верной. Термин "технология" в применении к программированию и отражал эту позицию".

Наш ответ на поставленный вопрос заключается в том, что программирование сочетает в себе и творческие проявления (как науку, так и искусство), и ремесло, понимаемое как профессионализм в утилитарном смысле, и технологию, позволяющую вести работы в промышленном масштабе. Другое дело, что их соотношение может быть очень разным в зависимости от конкретной задачи, конкретных программистов, избранных методологии, технологии, языка и системы программирования. Поэтому более корректно говорить о том, как наука, искусство, ремесло и технология могут сочетаться и сосуществовать при решении некоторой программистской задачи, некоторого проекта.

Конечно, в первую очередь наличие тех или иных проявлений определяется личными особенностями программиста. Две программы, написанные разными людьми для решения одной и той же задачи, нередко оказываются весьма непохожими друг на друга. Обратим внимание на то, что здесь нет четких критериев для оценки программ, как нет их и у искусствоведов для оценки художественного произведения. Святослав Сергеевич Лавров [Лавров 2000] отмечает, что "программирование часто становится мало похожим на искусство, опускаясь до уровня ремесла". Однако ремесленные по своей природе изделия в руках большого мастера приобретают высокую художественную ценность. Парадные портреты знатных людей писались мастерами как ремесленнические изделия, но стали произведениями искусства. Желательно, чтобы все проявления присутствовали в труде программиста в сочетании, которое подчеркивает лучшие стороны каждого из них.

Если говорить о большом проекте, в работе над которым участвует много программистов, то наличие тех или иных проявлений будет определяться типом решаемой задачи. Вот лишь некоторые примеры. Реальная творческая обстановка решения сложной задачи вносит элементы искусства. Рутинная, но требующая высокого качества задача будет тяготеть к элементам технологии. Если задача связана с необходимостью математического обоснования или верификации, то очевидна ее близость к науке. А решение интересной, но очередной однотипной задачи будет ремеслом.

Для каждой из задач можно построить профильный график, наподобие хорошо известной "розы ветров". Будем в северном направлении указывать близость

задачи к науке, в южном — к ремеслу, в западном — к технологии, а в восточном — к искусству. Тогда, например, для задачи разработки операционной системы, характер профильного графика может быть таким, как на рис. 1.6.

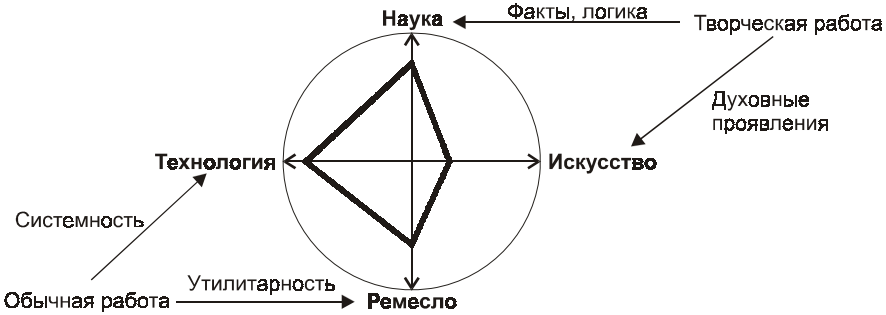


Рис. 1.6. Взаимосвязь программирования с искусством, наукой, ремеслом и технологией

Обратим внимание на то, что попытки исследовать творческую составляющую программирования предпринимались неоднократно. В статье "Программирование как высшая форма творчества" (**http://www.bugtraq.ru/library/misc/creative.html**) указывается на различия программирования и любой другой формы творчества.

- Программирование *не использует рефлексию в качестве метода познания.* Художник, композитор или писатель, созидая, решают собственные душевные проблемы. Такова мотивация всех видов творчества, но с программированием обычно все обстоит иначе.
- Программирование *не направлено на душу потребителя.* Программные продукты могут помочь пользователю в работе, развлечь, связать с другими людьми, но они, как правило, никак не скажутся на душевном состоянии.
- В программировании *практически отсутствует плагиат.* Программист может использовать огромное количество библиотек и фрагментов программ. Он реализует свой продукт, базируясь на чужих наработках.

Есть некоторые программы и фрагменты программ, которые следует воспринимать как настоящие произведения искусства. Их надо выставлять в музеях или, по крайней мере, в интернет-музеях, где все желающие могли бы любоваться ими и учиться программировать на примерах таких программ. В листинге 1.1 мы приводим одну программу для такого музея и призываем читателя насладиться ее красотой. Попробуйте сделать это самостоятельно, без помощи "искусствоведа"!

Программа представляет решение Г. С. Цейтина задачи расстановки ферзей на шахматной доске с хранением информации в локальных процедурах. Напомним, что в этой задаче требуется составить программу, генерирующую

все расстановки восьми ферзей на шахматной доске из 8×8 полей с таким условием, чтобы ни один ферзь не мог бить никакого другого ферзя. Программа написана на языке Algol-68.

Листинг 1.1. Задача расстановки ферзей на шахматной доске

```
proc Ферзи = (int k, proc(int, int) bool Можно, proc void Печать) void:
    if k = 8 then
       Печать
    else
       int p = k+1;
       for q to 8 do
           if Можно(p, q) then

              proc МожноНовое = (int x,y) bool:
                  if y = q or x-p = abs(y-q) then
                     false
                  else
                     Можно(x, y)
                  fi;

              proc ПечатьНовое = void:
                  (Печать; print((",", "abcdefgh"[p], "12345678"[q])));

              Ферзи(p, МожноНовое, ПечатьНовое)
           fi
       od
    fi;

proc МожноНачальное = (int x, y) bool: true;
proc ПечатьНачальная = void: print(newline);

Ферзи(0, МожноНачальное, ПечатьНачальная);
```

После того, как Вы разберетесь в алгоритме и идеях и оцените высокое искусство и авангардизм данной программы, обратите Ваше внимание на "оборотную сторону медали". Если некоторая часть большой системы будет написана талантливым программистом в таком творческом стиле, то после его увольнения при необходимости сопровождения написанного кода, возможно, будет легче переписать его заново.

Кстати, на задачу расстановки ферзей обращали внимание многие великие программисты. Рекомендуем посмотреть вариант программы, предложенный Дейкстрой [Дейкстра 1975].

1.6. Кризис программирования

Когда дело доходит до самооправдания, тут все мы гении.
З. Фрейд

Вот проявления так называемого *кризиса программирования*.

- Большие проекты выполняются с отставанием от графика.
- Большие проекты выполняются с превышением сметы расходов.
- Разработанный продукт не обладает требуемыми функциональными возможностями.
- Производительность разработанного продукта низка.
- Качество разработанного продукта не устраивает пользователя.

Эти симптомы хорошо известны всем программистам, участвовавшим в достаточно больших проектах. Брукс [Брукс 1999] указывает в качестве основных причин существования кризиса программирования четыре *присущих программному обеспечению свойства*.

- *Сложность*. Оказалось, что много проблем проистекает из человеческой природы и человеческой неспособности обрабатывать сложность.
  - Сложность программ является существенным, а не второстепенным свойством. Поэтому описания программных объектов, абстрагирующиеся от их сложности, часто абстрагируются и от их сущности.
  - Сложность лежит в основе административных проблем. Трудность осуществления надзора влечет ослабление концептуальной целостности.
- *Согласованность*. Значительная часть сложности происходит от согласования с другими интерфейсами.
- *Изменяемость*. Программное обеспечение очень легко изменить. При этом появление добавленного кода ("заплат") может привести к разрушению идей, заложенных в код изначально.
- *Незримость*. Реальность программного обеспечения не встраивается естественным образом в пространство. У программного обеспечения сверхсложное геометрическое представление: как правило, большое количество неориентированных графов, наложенных один на другой.

Важнейшие причины кризиса связаны с понятием "*человеческий фактор*". Одной из первых серьезных работ в этой области стала книга "Психология программирования" [Weinberg 1998], изданная еще в 1971 г. Все большее количество специалистов считают людей, участвующих в проектах, наиболее важным фактором и наиболее важным компонентом в создании программных продуктов.

Именно люди определяют успешность или неуспешность проекта. Приведем лишь некоторые причины, обратив внимание на их очевидность.

- Программисты обладают личными свойствами характера, они *изменчивы и непостоянны* (некоторые авторы называют это нелинейностью).
- *Невозможно проанализировать и обобщить действия великих программистов за работой.* Давно возникло понимание того, что некоторые программисты на порядок, а то и на несколько порядков более полезны, чем остальные. К сожалению, немногочисленные объяснения причин такого состояния по-прежнему остаются лишь на уровне гипотез.
- Методологии, технологии и языки программирования практически *не учитывают особенности человеческой натуры.*

Многие специалисты считают, что в основе кризиса лежат *неоптимальные управленческие решения.* Отметим несколько причин, связанных с менеджментом программного проекта.

- *Неумение менеджера правильно сформировать проектную команду.* Эдвард Йордон (Edward Yourdon) [Йордон 2001] отмечает как минимум две составляющих решения проблемы. Первая — становление открытых и честных взаимоотношений в команде. Вторая — подбор психологически совместимых исполнителей.
- *Непонимание начинающими менеджерами простейших вещей.* Например, поскольку "все программисты — оптимисты", нельзя доверять срокам, называемым программистами.
- *Отсутствие нормальных условий работы.* Том ДеМарко (Tom DeMarco) и Тим Листер (Tim Lister) [DeMarco Lister 1987] установили, что продуктивность работы тех, кто находится в хорошем офисе и может, закрыв дверь, не отвлекаться на телефонные звонки и посторонние дела, почти в 2.6 раза выше, чем у находящихся в коллективных комнатах.

Следствием конкуренции на современном рынке программного обеспечения можно объяснить еще одну важную причину кризиса программирования — *экстремальные условия*, в которых выполняются многие проекты. Йордон [Йордон 2001] вводит понятие "безнадежного проекта", т. е. проекта, хотя бы один параметр которого значительно отклоняется от нормальных значений. Перечислим эти параметры.

- План проекта сжат более чем на 50% по сравнению с нормальным расчетным планом. Жесткая конкуренция делает такую ситуацию очень распространенной.
- Число разработчиков уменьшено более чем на 50% по сравнению с действительно необходимым числом для выполнения проекта данного размера и масштаба.
- Бюджет и связанные с ним ресурсы урезаны более чем на 50%. Это может быть и следствием сокращения компании и следствием борьбы за выгодный контракт, подписанный на жестких финансовых условиях.

- Требования к функциям, возможностям, производительности и другим техническим характеристикам более чем на 50% превышают значения, которые они могли бы иметь в нормальных условиях. Например, таким требованием может быть написание компилятора, скорость компиляции которого должна быть вдвое больше по сравнению с любым известным компилятором для данной архитектуры.

Косвенное влияние на кризис оказывает и то, что во многих областях программирования существует серьезный *разрыв между теорией и практикой*. Первые три тезиса взяты из работы [Glass 1996].

- Практика значительно опережает теорию в области проектирования программного обеспечения.
- Область сопровождения программ долгое время игнорировалась теоретиками, однако серьезный теоретический рывок здесь был сделан за последнее десятилетие.
- Совершенно разными дорогами идут теория и практика в области разработки метрик программного обеспечения. Несмотря на то, что теоретики предложили значительное количество метрик, практики используют их очень редко, а если и используют, то собственные.
- Обратим внимание и на незначительные результаты исследовательской работы в области разработки программного обеспечения. Исследования, проводимые в последнее время в университетах и в большинстве коммерческих компаний, являются фрагментарными и односторонними. Многие крупные компании славятся не столько генерированием новых идей, сколько доводкой заимствованных или купленных.

1.7. Основные понятия профессионального программирования

Определение понятий — это начало ссылок.
Сократ

В этом разделе мы сначала поясним необходимость введения понятий и классификаций, а затем подробно поговорим о нескольких основных понятиях, которые потребуются нам в остальных главах этой книги.

1.7.1. Отступление "о понятиях, определениях и классификациях"

Не забывай меня, казни меня,
Но дай мне имя, дай мне имя!
Мне будет легче с ним, пойми меня,
В беременной глубокой сини!
О. Э. Мандельштам

Существует шуточное утверждение, согласно которому определения нужны "для того, чтобы о том, что определяется, можно было сказать в дискуссии

хоть что-нибудь определенное, а сама дискуссия не превращалась в бесплодный спор ослов о словах" [Шевчук, Харахоркина, Журавлев, Армен 1998]. На самом деле, понятия очень важны с точки зрения теории познания и логики. Кибернетическое "понятие понятия" дано в книге [Турчин 2000].

Понятие — множество ситуаций на входе кибернетической системы. Владеть понятием — это значит уметь его распознавать, т. е. уметь определять, принадлежит ли любая данная ситуация к множеству, характеризующему понятие, или не принадлежит. Проще говоря, понятие — это более или менее четко распознаваемое множество ситуаций или иных объектов.

Определение — логическая операция, заключающаяся в придании точного смысла языковому выражению. Придание точного смысла сводится к созданию метода распознавания принадлежности объектов множеству. Цель определения — уточнить содержание используемых понятий.

> ***О русскоязычной и англоязычной терминологиях***
>
> Следует отметить, что англоязычная терминология начала проникать в лексикон программистов постепенно и не сразу. Это косвенно свидетельствует о том, что в области вычислительного дела поначалу никакой зависимости СССР от Запада не было. Лишь позже "ЭВМ" превратилась в "компьютер", "программирующая программа" в "транслятор" и "компилятор", а "автокод" в "ассемблер". Далее появились "дисплеи", "утилиты" и многое другое, что составляет терминологию, с трудом понимаемую программистами первого поколения.

Классификация — система, согласно которой что-либо распределяется по группам, разрядам, признакам, принципам, классам. Навести порядок в хаосе очень важно для того, чтобы лучше понимать исследуемую предметную область.

Одна из наиболее известных и удачных классификаций — периодический закон и периодическая система химических элементов Дмитрия Ивановича Менделеева. Менделееву удалось найти такое расположение химических элементов, благодаря которому появилась возможность получить четкую картину соотношения элементов и предсказать открытие новых элементов.

Если же классификационные принципы выбраны наугад, то мы получим нечто подобное следующей классификации, описанной аргентинским писателем Хорхе Луисом Борхесом в новелле "Аналитический язык Джона Уилкинса": "Животные подразделяются на: а) принадлежащих Императору, б) бальзамированных, в) прирученных, г) молочных поросят, д) сирен, е) сказочных, ж) бродячих собак, з) включенных в настоящую классификацию, и) буйствующих, как в безумии, к) неисчислимых, л) нарисованных очень тонкой кисточкой из верблюжьей шерсти, м) и прочих, н) только что разбивших кувшин, о) издалека кажущихся мухами".

Напоследок обратим внимание, что многие определения и классификации субъективны (как и все, к чему причастен человек). Тем не менее, чтобы

вести обсуждение какого-либо предмета, следует определить его и указать его место в картине мира. Надеемся, что область наших исследований будет сформулирована достаточно четко. А если кого-то и не устроят приведенные в этой книге определения, заметим, что даже для термина "математическая статистика" (наука, которой более 100 лет) существует коллекция из более 200 определений [Налимов 1979].

1.7.2. Программа и программный продукт

В основе двух следующих определений лежат идеи, принадлежащие Фредерику Бруксу [Брукс 1999]. Они отражают разницу между программой и программным продуктом, являющуюся достаточно простой. Программа пишется "для себя", а программный продукт создается "для заказчика" и предполагает значительно более серьезный подход к его созданию и сопровождению.

Программа (program) — завершенное изделие, пригодное для запуска своим автором на системе, на которой оно было разработано. Набор взаимодействующих программ, согласованных по функциям и форматам, точно определенным интерфейсам и вкупе составляющих полное изделие для решения больших задач, образует *программный комплекс*.

Программный продукт (program product) — программа или программный комплекс, предназначенный для продажи или передачи в эксплуатацию другим лицам и удовлетворяющий ряду требований. Программный продукт должен соответствовать единому стилю программирования, быть тщательно оттестирован, поставляться с подробной документацией и конфигурационными настройками, а также должен сопровождаться.

> ***Об определении программного продукта***
>
> Брукс определял программный продукт как программу, которую "любой человек может запускать, тестировать, исправлять и развивать". В этом определении не учтена коммерческая составляющая программного продукта. Под словосочетанием "любой человек" в случае коммерческого программного обеспечения будем понимать программиста, имеющего разрешение работать с исходными текстами программ.

Два следующих понятия отражают другую сторону программы — отличие ее от физических, "осязаемых", "окаменевших" частей вычислительной системы (аппаратных средств и аппаратного обеспечения).

Программное средство (software) — программа или программный комплекс на носителях данных, снабженный программной документацией [Жоголев 1996].

Программное обеспечение (software) — совокупность программных средств, входящих в состав вычислительной системы. Цель этих средств — обеспечение

эффективной работы ЭВМ и предоставление пользователю определенных видов обслуживания. Различают два основных типа программного обеспечения.

- *Системное программное обеспечение* является необходимым дополнением к аппаратному обеспечению. Важнейший компонент системного программного обеспечения — операционная система.
- *Прикладное программное обеспечение* состоит из прикладных программ, специфика которых определяется ролью ЭВМ и потребностями пользователя. Примерами могут служить пакеты прикладных программ и системы управления базами данных.

О разделении на аппаратное и программное обеспечение

Конечно, аппаратное и программное обеспечение логически эквивалентны. Любая операция, которая может быть выполнена программно, может быть выполнена и аппаратно. Другое дело, что в подавляющем большинстве случаев это просто экономически невыгодно.

1.7.3. Алгоритмы

Если "алгоритм", то "аритметика".
Программистский фольклор

Алгоритм — точное и конечное описание того или иного общего метода, основанного на применении исполнимых элементарных тактов обработки.

Компьютер — *вычислитель*, он не понимает программу, а исполняет ее. Наиболее естественный способ указать компьютеру ход исполнения программы — записать ее в виде алгоритма (на алгоритмическом языке). Современное значение слова "алгоритм" во многом аналогично таким понятиям, как рецепт, процесс, метод, способ. Алгоритм имеет пять важных свойств [Кнут 2000].

- *Конечность.* Алгоритм всегда должен заканчиваться после выполнения конечного числа шагов.
- *Определенность* (*детерминированность*). Каждый шаг алгоритма должен быть точно определен.
- *Наличие входных данных.* Алгоритм имеет некоторое число входных данных, задающихся до начала его работы или определяющихся динамически во время его выполнения.
- *Наличие выходных данных.* Алгоритм имеет одно или несколько выходных данных, имеющих определенную связь с входными данными.

- *Эффективность.* Алгоритм обычно считается эффективным, если его операторы достаточно просты для того, чтобы их можно было точно выполнить в течение конечного промежутка времени.

С алгоритмами связаны следующие области исследований.

- *Анализ алгоритмов.* Предмет этой области состоит в том, чтобы для заданного алгоритма определить рабочие характеристики. Например, часто требуется, чтобы алгоритм был быстрым.
- *Теория алгоритмов.* К этой области относятся вопросы существования или не существования эффективных алгоритмов вычисления определенных величин.
- *Построение алгоритмов.* В этой области рассматриваются стандартные приемы и методы, используемые при написании алгоритмов.

Большинство используемых на практике алгоритмов, с которыми работают программисты, являются полиномиальными. Это означает, что время работы алгоритма на входе длины `n` составляет `O(n)`. Однако не всякая задача может быть решена за полиномиальное время. Некоторые решаются лишь за экспоненциальное, а некоторые вообще не могут быть решены никаким алгоритмом.

Имеется особый класс задач, называемый *"NP-полные" задачи.* Для этих задач не известны полиномиальные алгоритмы, однако и не было доказано, что таких алгоритмов не существует. Для программиста знание о NP-полных задачах важно по следующей причине. Если для некоторой задачи удалось доказать, что она NP-полная, то есть основания считать ее практически неразрешимой. В этом случае лучше потратить время на построение приближенного алгоритма, чем продолжать искать быстрый алгоритм, решающий ее точно.

Профессионального программиста обычно в первую очередь интересуют два вопроса, связанных с алгоритмами.

- *Где найти справочник по важнейшим классам быстрых алгоритмов?*

  В этом случае можно порекомендовать четыре книги.

  - "Искусство программирования, том 1. Основные алгоритмы" [Кнут 2000].
  - "Алгоритмы: построение и анализ" [Кормен, Лейзерсон, Ривест 1999].
  - "Алгебраическая алгоритмика" [Ноден, Китте 1999].
  - "Алгоритмы и структуры данных" [Вирт 2001].

- *Какова цена человеческих усилий и за какое время алгоритм решает задачу?*

  Еще раз подчеркнем, что алгоритмически разрешимых задач хоть и бесконечно много, но сравнительно мало — всего счетное множество. Да и среди алгоритмически разрешимых большинство задач экспоненциальной

сложности. Для тех, кто заинтересуется этим вопросом, кроме [Кнут 2000], порекомендуем две книги.

- "Основы теории элементарных алгоритмов" [Косовский 1987].
- "Теория алгоритмов: основные открытия и приложения" [Успенский, Семенов 1987].

О написании слова алгоритм

"Санкт-Петербургская — Ленинградская" школа логики исторически использует другое написание этого слова — алгорифм. Оба написания являются одинаково правильными. Отдавая дань уважения школе, например, пишут "нормальные алгорифмы Маркова" (это также позволяет избежать противопоставления нормальных алгоритмов ненормальным).

1.7.4. Модели и моделирование

Придумать зеленое солнце легко; трудно создать мир,
в котором оно было бы естественным...

Дж. Р. Р. Толкиен

Модель — это упрощенное представление реальности. Моделью является, например, чертеж системы. Основное свойство модели в том, что она — семантически замкнутая абстракция системы. Она строится для того, чтобы лучше понять разрабатываемую систему, визуализировать ее, определить структуру или поведение. Сложные системы моделировать просто необходимо, поскольку иначе мы не сможем их воспринять как единое целое. Разные модели по-разному отражают структуру системы, а поэтому ни одна из них не определяет ее.

Моделирование — метод исследования систем на основе переноса изучаемых свойств системы на объекты другой природы. Главное в моделировании — отношение подобия между объектом моделирования и его моделью. Это один из основных методов исследования окружающей действительности. Инженерная методика изготовления моделей является устоявшейся и повсеместно принятой. Она позволяет решить несколько важных задач:

- визуализировать систему;
- определить структуру системы и ее поведение;
- документировать принимаемые решения.

Наконец, моделирование — это попытка решить проблему сложности.

Существует четыре основных принципа моделирования [Буч, Рамбо, Джекобсон 2000].

- Выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение.
- Каждая модель может быть воплощена с разной степенью абстракции. При этом переход от одной степени абстракции к другой дает нам новую модель.

- Лучшими моделями являются те, которые ближе к реальности (по некоторому выбранному критерию).
- Следует использовать совокупность нескольких моделей.

> ***О моделировании при разработке программ***
>
> Собственно саму разработку программ мы можем рассматривать как последовательную трансформацию моделей, двигаясь от периферии к ядру, — от модели спецификаций до алгоритмической модели.

Выделяют три основных вида моделей в зависимости от их сущности. Это натурные модели (например, макеты), воображаемые (например, геометрическая точка) и информационные (например, программа). Информационные модели, наиболее интересные нам, образуют следующие типы.

- *Дескриптивные модели*, выраженные на языке описания. Это может быть естественный язык или специальный: научный (для алгоритмов) и технический (для программ).
- *Наглядные модели*, выраженные на языке представления. Например, рисунки, чертежи или фотографии.
- *Смешанные модели*, использующие как язык описания, так и представления. Например, это могут быть диаграммы, схемы, таблицы, графы.

1.7.5. Жизненный цикл программного обеспечения

Он пришел к реке. Река была на месте.
Э. Хемингуэй. "Большая река двух сердец"

Жизненный цикл программного обеспечения — весь период его разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращением всех видов его использования.

Существует простейшее представление жизненного цикла, которое включает следующие стадии:

- анализ;
- проектирование;
- программирование;
- тестирование и отладка;
- сопровождение и эксплуатация.

Цикл разработки программного обеспечения тесно связан с технологиями программирования, поэтому подробно его мы рассмотрим в *гл. 3*, посвященной технологиям.

О цикле

Традиционно мы говорим "жизненный цикл", подчеркивая повторяемость стадий. Действительно, за исключением простейшего случая, разработка имеет итеративный характер, возвращающий к одному из предыдущих этапов.

1.7.6. Проект и команда

Два важнейших и очень тесно связанных друг с другом понятия профессиональной разработки — "проект" и "команда". Проект реализуется командами, а команда работает над проектами.

1.7.6.1. Проект

Начнем с определения этого понятия. *Проект* (project) — это ориентированное на программный продукт объединение действий разработчиков. Подчеркнем, что действие ориентировано именно на продукт, а не на отдельный процесс (проектирование, тестирование и т. п.).

Рассмотрим три основных *категории разработок* и соответствующие им проекты и проиллюстрируем их примерами из области разработки компиляторов.

- *Свойство* (feature) программного продукта — архитектурный элемент, направленный на некоторый специфический аспект продукта. Свойство может быть реализовано в одном или в нескольких компонентах. Примером свойства является система генерации информации для отладчика в языково-зависимой части компилятора.
- *Компонент* (component) — совокупность свойств. Обычно компонент предполагает значительность, и за нее несет ответственность команда разработчиков компонента. Примеры компонентов — языково-зависимая часть компилятора с языков Pascal или FORTRAN.
- *Продукт* (product) — совокупность компонентов, объединенная с целью решения определенных задач пользователя. Над продуктом работает команда разработчиков продукта. Для коммерческих разработок продукты включены в каталог продуктов компании. Типичные примеры продукта — Pascal-компилятор или FORTRAN-компилятор. Результатом *проекта продукта* будет конкретная версия (например, 2.0) Pascal-компилятора.

Существует еще одна категория проектов.

Исследовательский проект — проект, который не утвержден для включения в состав какой-либо версии. Для коммерческих разработок это означает также отсутствие полноценного финансирования. Обычно это зарождающийся проект, однако он может никогда не вырасти в полноценный проект.

Об исследовательских проектах

Грамотный и умный заказчик не может не финансировать исследовательскую деятельность. Конечно, данное утверждение не всегда соответствует жизненным реалиям, но на него следует ориентироваться.

Рассмотрим также некоторые характеристики проектов.

- Проект имеет *специфику, определяемую типом заказчика*. Наиболее существенными отличиями в процессе разработки будут отличия в процессах управления, возникновения и исследования идеи. Проект может находиться в диапазоне, граничными значениями которого являются:
  - проект *для широкого круга пользователей*;
  - проект *для конкретного заказчика*.

О промежуточных вариантах

Кроме широкого круга пользователей и конкретного заказчика бывают интересные промежуточные варианты. Из некоторого большого проекта можно сделать частное решение для определенного клиента. В большой проект может быть добавлена часть, нужная одному-единственному, но очень важному клиенту. Наконец, может существовать несколько версий одного и того же продукта, ориентированных на различные целевые группы пользователей.

- Существуют граничные значения для проектов, определяемые *степенью коммерциализации*:
  - *коммерческий* проект. Это проект, разработчики которого ориентируются на получение прибыли от продажи программного продукта;
  - *некоммерческий* проект. Такие проекты обычно являются нерентабельными. Мотивация для участия в таких проектах — приобретение опыта в конкретной области программирования, приобретение статуса и уважения, обучение работе в коллективе. Некоммерческий проект может иметь особые, уникальные черты. Очень часто такие проекты являются "общинными" проектами (**http://www.osp.ru/os/1999/09-10/071.htm**). Они характеризуются децентрализацией. Менеджер (координатор) проекта практически все время выполнения проекта не знает точного количества участников. Практически всегда это разработка с открытыми и доступными всем текстами программного продукта. Как правило, некоммерческие разработки распространяются с лицензией, запрещающей их использование в коммерческих целях.
- Еще одной характеристикой проекта является его *масштаб*. В зависимости от масштаба можно выделить четыре категории проектов [Йордон 2001]:
  - *небольшой проект*. Команда разработчиков включает менее 10 человек. Протяженность проекта от 3 до 6 месяцев;
  - *средний проект*. Команда разработчиков включает от 20 до 30 человек. Протяженность проекта от 1 до 2 лет;

- *крупномасштабный проект.* Команда (иерархия команд разработчиков) включает от 100 до 300 человек. Протяженность проекта от 3 до 5 лет;
- *гигантский проект.* Команда (иерархия команд разработчиков) включает от 1000 до 2000 человек. Протяженность проекта от 7 до 10 лет.

Мы еще раз вернемся к понятию проекта при обсуждении процесса управления в *разд. 3.2.2.*

1.7.6.2. Участники процесса разработки

Основные действующие группы можно представить следующим перечислением: пользователи, управляющий комитет, несколько советов консультативного характера и команды разработчиков. Информационные потоки между основными группами представлены на рис. 1.7 и могут быть прокомментированы следующим образом:

- управляющий комитет получает планы от команд и утверждает (или не утверждает) их;
- команда разработчиков продукта объединяет коды нескольких компонентов, составляющих проект продукта. В некоторый момент от основных кодов ответвляется очередная версия для пользователей, за которую несет ответственность команда версии;
- пользователи (заказчики) выдвигают требования и получают прототип от команды разработчиков компонента, а затем различные версии продукта от команды версии.

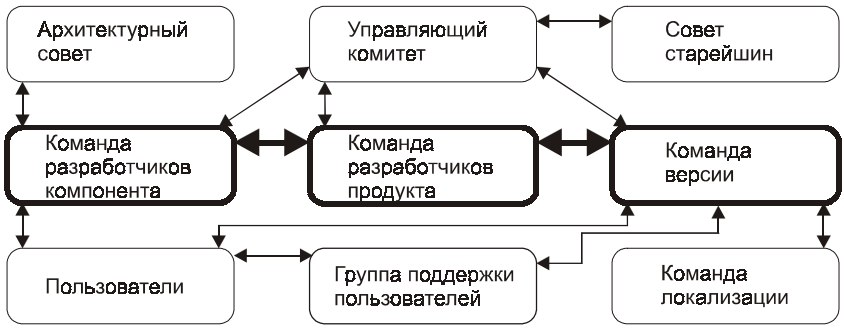


Рис. 1.7. Участники процесса разработки

Рассмотрим роль отдельных действующих групп более подробно.

- *Пользователи* (*заказчики*) являются главными действующими лицами, вовлеченными в процесс разработки. Собственно, если нет пользователей, то обычно нет и продукта. Крупные компании, заинтересованные в

увеличении количества пользователей, всячески приветствуют контакты с ними, включая визиты в компанию и образование пользовательских групп.

- *Управляющий комитет* разрабатывает стратегию и направления развития, утверждает планы, определяет необходимость выпуска очередной версии продукта.
- *Архитектурный совет* обеспечивает информационный и технический базис для параллельного развития продуктов и следит за архитектурным постоянством изменений.
- *Совет старейшин* состоит из наиболее опытных и уважаемых сотрудников и привлекается управляющим комитетом с целью распознавания возможных проблем и оценки перспектив.
- *Группа поддержки пользователей* оперативно отвечает на сложные технические вопросы клиентов, собирает жалобы и выпускает "заплаты" к программным продуктам.

Существует два типа команд разработчиков.

- *Команда разработчиков продукта* начинает, исследует и развивает проект продукта. Лидером команды является менеджер проекта программного продукта. Эта команда состоит из представителей всех функциональных подразделений, которые занимаются разработкой совместно:
  - инженеров-разработчиков (специалистов по инженерии программирования и программистов);
  - технических писателей;
  - инженеров тестирования;
  - инженеров качества;
  - специалистов по сопровождению продукта;
  - специалистов по продажам продукта.

  Команда полностью проектирует и разрабатывает продукт. Обычно команда разработчиков продукта включает в себя несколько команд разработчиков компонентов.
- *Команда разработчиков компонента* предназначена для создания и развития компонента или свойства. Команду возглавляет обычно менеджер проекта компонента, а входят в нее:
  - инженеры-разработчики;
  - инженеры тестирования;
  - технические писатели.

Команда начинает работу сразу после завершения проектирования продукта. Именно эта команда после начала продаж продукта занимается сопровождением, исправляя ошибки, обнаруженные пользователем.

Существуют также команды, ориентированные на выпуск версии.

- *Команда версии* отвечает за производство и помещение альфа-версии, бета-версии и версии первой поставки пользователю на страницах сервера Интернета, дискетах, компакт-дисках или других носителях. Версия может включать один или более продуктов. Команда версии состоит из представителей команд разработчиков и возглавляется менеджером версии программного продукта. Команда отслеживает и управляет графиком разработки и поставок, принимает решения относительно объединения продуктов и компоновки пакетов на выбранном носителе. Обычно версия и проект имеют некоторые символические названия (например, географические: "Ладога" или "Петергоф").
- *Команда локализации* решает проблемы локализации (адаптации) продукта для конкретной языковой и культурной среды. При этом осуществляется перевод сообщений программы и документации к ней на язык локализации продукта.

Собственно последовательность версий программного продукта может быть проиллюстрирована следующим образом (рис. 1.8). Существует одна основная эволюционирующая версия, от которой по мере готовности ответвляются версии, предназначенные для пользователей. Если новая версия включает существенные изменения (например, новый интерфейс), то увеличивается на единицу ее "основной" номер (число перед точкой). Если изменения незначительны и сводятся к небольшим поправкам функциональности, то увеличивается на единицу "второстепенный" номер (число после точки). В ответвившиеся версии также могут вноситься исправления (обычно только исправления ошибок). Только основная версия развивается и может содержать кардинальные изменения, остальные версии лишь сопровождаются.

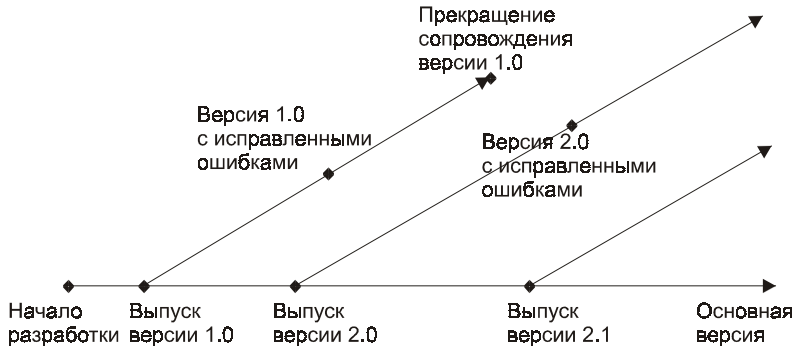


Рис. 1.8. Последовательность версий программного продукта

1.7.7. Платформы

Нам не хватает практики, а не правил.
Карл Вигерт

В этом разделе мы обсудим одно из понятий, которое стало очень популярным в последнее время. На самом деле оно близко другому понятию, имеющему длинную историю и входящему в классические учебники. Давно известен подход, называемый *многоуровневой компьютерной организацией*. Благодаря построению ряда уровней абстракций, каждая из которых надстраивается над абстракцией более низкого уровня, подход позволяет решить сложности, возникающие при общении с компьютером. В результате, в самом простейшем случае возникает трехуровневая организация (архитектура) ЭВМ (рис. 1.9), состоящая из аппаратного, операционного и языкового уровней.

Рис. 1.9. Трехуровневая организация (архитектура) ЭВМ

Платформа (platform) — видимые программисту средства поддержки программного продукта каждым из этих уровней абстракции. Определяют следующие типы платформ.

- *Аппаратная платформа* — тип аппаратной архитектуры, на котором может быть установлен программный продукт. Наиболее известны аппаратные платформы на семействах процессоров Intel, HP и Power.
- *Операционная платформа* — тип операционной системы и системного интерфейса, с которым может работать программный продукт. Наиболее известные операционные платформы: DOS, Windows и Unix. *Сетевая платформа* обычно входит в состав операционной платформы и представляет собой сетевой интерфейс операционной платформы. Частью операционной платформы являются библиотеки, поставляемые с операционной системой, для работы с которыми используется интерфейс прикладного программиста.

- *Языковая платформа* — язык программирования и интерфейс прикладного программирования (библиотеки), на котором может быть реализован программный продукт. Наиболее известные языковые платформы: C, C++, Java.

Об инструментальной платформе

Инструментальная платформа (система программирования) в данном контексте может рассматриваться как продолжение операционной. Граница между ними в ряде случаев весьма условна. Инструментальная платформа обычно содержит библиотеки, для работы с которыми используется интерфейс прикладного программиста.

Обратим внимание на то, что пользователь программного продукта, как правило, должен представлять, для какой аппаратной и операционной платформ продукт предназначен.

Интерфейс прикладного программиста (библиотеки) может иметь и операционная, и инструментальная, и языковая платформы. Кроме того, одна и та же платформа может выпускаться в разных редакциях (изданиях), имеющих различные наборы возможностей по интерфейсу прикладного программиста. Для платформы Java, например, это три следующих редакции:

- *корпоративная* (enterprise) редакция, для разработки и развития многоуровневых приложений уровня предприятия и управляемых центральным сервером;
- *стандартная* (standard) редакция, предназначенная для разработки, развития и управления межплатформенными универсальными приложениями;
- *микро* (micro) редакция, предназначенная для малых автономных потребителей, с целью разработки, развития и управления приложений, варьирующихся от смарт-карт до обычных вычислительных устройств.

Могут возникать устойчивые совокупности платформ, являющиеся, как правило, взаимовыгодными для всех входящих в совокупность платформ. Так, в начале 1990-х годов сформировалась аппаратно-операционная платформа, получившая имя Wintel. В этом названии отражены названия платформ двух компаний: Microsoft и Intel. Еще один пример — это Java-платформа (или, например, более конкретно J2EE-платформы) (**http://ru.sun.com/win/java/**) от компании Sun Microsystems и .NET-платформа (**http://www.microsoft.com/rus/net/**) от компании Microsoft, конкурирующие между собой. Причем, если некоторое время назад об этих платформах обычно говорили, как об инструментально-языковых, то сейчас более корректно говорить о них, как об операционно-инструментально-языковых.

О конкуренции платформ

Практически все компьютерные издания с интересом обсуждают вопросы конкуренции платформ. Скорее всего, не имеет смысла говорить о грядущей полной и безоговорочной победе одной из них. Каждая имеет и свои плюсы, и свои минусы. Каждая будет благополучно кормить и отдельных программистов, и целые компании. Причем как разрабатывающие программные продукты для каждой из платформ, так и интегрирующие или конвертирующие программные продукты с одной платформы на другую.

1.8. Основная задача профессионального программирования

Знай, читатель, что мудрость уменьшает жалобы,
а не страдания!
Козьма Прутков

Сформулируем основную *задачу профессионального программирования* как создание высококачественного программного обеспечения. Образно говоря, стержневой подзадачей является осуществление последовательной трансформации моделей при движении от периферии к ядру. Начинаем мы с первой модели — общей постановки задачи в предметной области и более конкретных моделей спецификаций — и заканчиваем алгоритмической моделью и моделью на языке машинных команд (рис. 1.10). Заметим, что *целью программирования* является получение результатов работы программы, а не создание красивых моделей или кодирование алгоритмов.

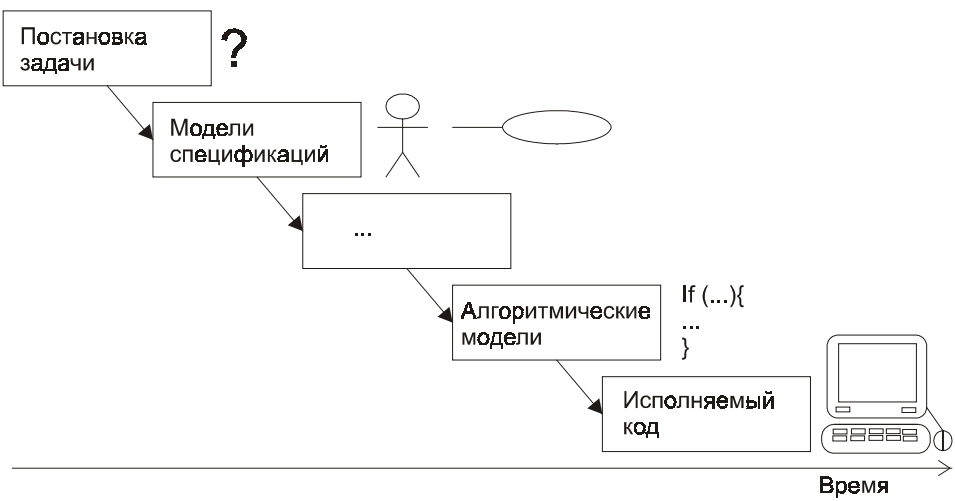


Рис. 1.10. Путь от постановки задачи к исполняемому коду

Основная задача программирования решается на профессиональном уровне с помощью тех основных направлений, которым и посвящена данная книга. Краткие рекомендации по их применению выглядят так:

1. Сначала следует определить методологию программирования, которая будет включать совокупность методов и концепций, объединенных общим философским подходом. Методология должна быть частью общей культуры профессионального программиста. Ее выбор осуществляется в самом начале работы над конкретным проектом (в зависимости от специфики задачи), и по ходу реализации проекта она не должна изменяться.
2. Далее следует выбрать технологический подход, который будет определять совокупность процессов, применяемых при разработке программного продукта. Определенная ранее методология включает совокупность методов, которые будут применены в технологическом подходе. Технологический подход может изменяться в ходе работы над проектом и приспосабливаться к его нуждам.
3. Методология и технология определяют языки и системы программирования, необходимые для каждого процесса избранного технологического подхода.
4. Технологические процессы будут исполняться на некоторых аппаратной и операционной платформах. Заметим, что аппаратная и операционная платформы могут существенно определять наличие и специфику инструментов (систем программирования). В большинстве разработок следует избегать зависимости от платформ, однако ряд проектов (как правило, системных) в большой степени опирается на их хорошее знание.

Удачи!

ЛИТЕРАТУРА

Достоевский дает мне больше, чем любой мыслитель ...

Альберт Эйнштейн

1. *Брукс 1999* — Брукс Ф. Мифический человеко-месяц или как создаются программные системы. — СПб.: Символ-Плюс, 1999.
2. *Буч, Рамбо, Джекобсон 2000* — Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. — М.: ДМК, 1998.
3. *Вирт 2001* — Вирт Н. Алгоритмы и структуры данных. — СПб: Невский диалект, 2001.
4. *Гамма, Хелм, Джонсон, Влиссидес 2001* — Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного программирования. Паттерны проектирования. — СПб.: Питер, 2001.

5. *ГОС 220400* — Государственный образовательный стандарт высшего профессионального образования. Специальность 220400 — Программное обеспечение вычислительной техники и автоматизированных систем.

6. *Грис 1984* — Грис Д. Наука программирования. — М.: Мир, 1984.

7. *Грэхем, Кнут, Паташник 1998* — Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики. — М.: Мир, 1998.

8. *Дейкстра 1975* — Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975, с. 7—97.

9. *Достоевский 1985* — Достоевский Ф. М. Преступление и наказание. — М.: Мир, 1985.

10. *Ершов 1977* — Ершов А. П. Введение в теоретическое программирование. — М.: Наука, 1977.

11. *Йордон 2001* — Эдвард Йордон. Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте. — М.: ЛОРИ, 2001.

12. *Кнут 1974* — Дональд Кнут. Программирование как искусство // Лекции лауреатов премии Тьюринга. 1966—1985. — М.: Мир, 1993.

13. *Кнут 2000* — Дональд Кнут. Искусство программирования, том 1. Основные алгоритмы (3-е изд.). — М.: Издательский дом "Вильямс", 2000.

14. *Кормен, Лейзерсон, Ривест 1999* — Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999.

15. *Косовский 1987* — Косовский Н. К. Основы теории элементарных алгоритмов. — Л.: Изд-во Ленингр. ун-та, 1987.

16. *Лавров 2000* — Лавров С. С. Так что же такое информатика? // Компьютерные инструменты в образовании. — СПб.: Изд-во ЦПО "Информатизация образования", 2000, № 1, с. 22—25.

17. *Лавров 2001* — Лавров С. С. Программирование. Математические основы, средства, теория. — СПб.: БХВ-Петербург, 2001.

18. *Лавров, Слисенко, Цейтин 1985* — Лавров С. С., Слисенко А. О., Цейтин Г. С. Проект учебного плана специальности: "информатика и системное программирование" // Микропроцессорные средства и системы. — 1985, № 4, с. 20—28.

19. *Липский 1988* — Липский В. Комбинаторика для программистов. — М.: Мир, 1988.

20. *Налимов 1979* — Налимов В. В. Вероятностная модель языка. О соотношении естественных и искусственных языков. — М.: Наука, 1979.

21. *Ноден, Китте 1999* — Ноден П., Китте К. Алгебраическая алгоритмика. — М.: Мир, 1999.

22. *Поттосин 1997* — Поттосин И. В. Программистская инженерия: содержание, мнения и тенденции // Программирование. — М.: Наука, 1997, № 4, с. 26—37.

23. *Романовский 1999* — Романовский И. В. Дискретный анализ. — СПб.: Невский диалект, 1999.

24. *Турчин 2000* — Турчин В. Ф. Феномен науки: кибернетический подход к эволюции. — М.: ЭТС, 2000.

25. *Успенский, Семенов 1987* — Успенский В. А., Семенов А. Л. Теория алгоритмов: основные открытия и приложения. — М.: Наука, 1987.

26. *Уэзерелл 1982* — Уэзерелл Ч. Этюды для программистов. — М.: Мир, 1982.

27. *Цейтин 1990* — Цейтин Г. С. О профессионализме в программировании. — СПб.: ЛГУ, математико-механический факультет, рукопись, 1989.

28. *Шевчук, Харахоркина, Журавлев, Армен 1998* — Шевчук О. Б., Харахоркина А. А., Журавлев Б. С., Армен В. А. Защищайтесь, сударь! Руководство по созданию и защите диссертаций. — СПб.: Политехника, 1998.

29. *Curriculum 1968* — ACM Curriculum Committee on Computer Science. Curriculum 68: Recommendations for the undergraduate program in computer science // CACM, March 1968. V. 11, N 3, pp. 151—197.

30. *DeMarco, Lister 1987* — DeMarco Tom, Lister Tim. Peopleware. Dorset House Publishing, 1987.

31. *Denning 1989* — Denning P., et al. Computing as a Discipline // CACM, January, 1989. V. 32, N 1, pp. 9—23.

32. *Glass 1996* — Robert L. Glass. The Relation Between Theory and Practice in Software Engineering // CACM, November 1996. V. 39, N 11, pp. 11—13.

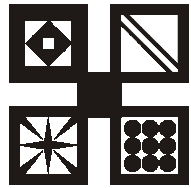
33. *Lethbridge 2000* — Timothy C. Lethbridge. Priorities for the education and training of software engineers. // The Journal of Systems and Software, 53, 2000, pp. 53—57.

34. *Leveson, Turner 1993* — Nancy G. Leveson, Clark S. Turner. An Investigation of the Therac-25 Accidents // Computer, July, 1993, pp. 18—41.

35. *Parnas 1990* — Parnas D. L. Education for Computing Professionals // Computer, January, 1990. V. 23, N 1, pp. 17—22.

36. *Tucker 1991* — Tucker A., et al. Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, N. Y. 1991.

37. *Weinberg 1998* — Gerald Weinberg. The Psychology of Computer Programming. Dorset House Publishing, 1988.



Глава 2

Методологии программирования

В этой главе рассказывается об основных методологиях программирования — совокупности методов, применяемых в жизненном цикле программного обеспечения и объединенных общим философским подходом. Основные вопросы главы следующие:

- Зачем нужны определяющие методологии в программировании?
- В чем заключается поддержка методологии методами и концепциями?
- Каковы вычислительные модели методологий?
- На какой класс практических задач ориентирована каждая из методологий?
- Какова языковая поддержка методологий?
- Каково происхождение методологий?

2.1. Введение в методологии программирования

2.1.1. Основные понятия и определения

Собрались однажды звери в лесу, чтобы строить мост через реку.
Когда все обсудили, медведь спрашивает: "Вопросы есть?"
Осел: "А как будем строить — вдоль или поперек реки?"

Анекдот о необходимости определяющих подходов

Мы определили *методологию* как совокупность методов, применяемых в жизненном цикле и объединенных общим философским подходом.

На сегодняшний день существует не так много методологий, особенно полных, т. е. учитывающих все стадии жизненного цикла программного обеспечения. Именно методология определяет, какие языки и системы будут применяться для разработки программного обеспечения и, во многом, рекомендует, какой технологический подход будет при этом использован.

С каждой методологией можно связать некоторые характерные для нее атрибуты.

- *Философский подход* (или основной принцип), являющийся простым для формулирования и определяющий основной источник эффективности методологии.
- Согласованное, связное множество моделей *методов*, через которые реализуется данная методология.
- *Концепции* (понятия, замыслы), поддерживающие методы и позволяющие более точно их определить.

Интересно также происхождение, история и эволюция методологии. Как правило, существует явный создатель, основоположник методологии. Любая методология создается на основе уже накопленных в предметной области эмпирических фактов и практических результатов. Для методологий программирования такими фактами и результатами являются уже существующие языки программирования.

О парадигмах программирования

Когда методология применяется во время стадии программирования (реализации), очень часто ее называют *парадигмой программирования* — способом мышления и программирования, не связанным с конкретным языком программирования. Термин "парадигма" был впервые предложен Томасом Куном [Кун 1977]. Он определял парадигму как свод норм научного мышления. Парадигма — это правило (modus operandi) развития научного знания. Оно в течение определенного времени дает научному сообществу модель постановки проблем и их решений.

Определенные методологии программирования поддерживаются определенными языками программирования. Это не означает, что несвойственные данной методологии языки совсем нельзя использовать. Однако Вам придется при этом потратить существенно большее количество усилий и ресурсов. Мы будем приводить краткое синтаксическое и семантическое описание языковой стороны методологий, а также указывать языки, поддерживающие методологии.

О синтаксисе и семантике мы еще поговорим в *гл. 4*, а сейчас приведем некоторые сведения, необходимые для понимания дальнейшего материала.

Синтаксис — правила образования текстов. Для описания синтаксиса далее будет использована расширенная формальная система обозначений Бэкуса-Наура. С ее помощью легко определить, какие именно последовательности символов будут считаться программами. В ней одни синтаксические категории определяются через другие последовательно. Метапеременные представляют собой слова или группы русских слов, заключенных в угловые скобки '<>'. Под значением метапеременной понимается некоторая конечная последовательность основных символов языка, из которых, в конечном счете, состоят программы. Символ '::=' означает "определяется как", символ

'|' означает "или". В случае произвольного количества повторений (в том числе ноля раз) некоторой конструкции будем заключать ее в фигурные скобки '{}'. Символы, указанные в квадратных скобках '[]', являются необязательными. Многие синтаксические определения в нотации Бэкуса-Наура рекурсивны. В таких случаях можно уточнить, что синтаксические категории определяются через другие лишь в конечном счете.

Семантика — правила истолкования текстов тем, кому они адресованы. В отличие от синтаксиса, семантика обычно описывается словесно, на естественном языке. Тем не менее, есть несколько подходов к формальному описанию семантики. Основным и исторически первым подходом является *операционный подход*. В нем семантика описывается в терминах некоторого вычислителя (машины), например абстрактной (воображаемой, виртуальной) машины. В процессе работы вычислителя меняется состояние программы, образуемое значениями ее переменных. Существует некоторое начальное состояние, а также результат исполнения программы.

Различные методологии программирования дают разный выигрыш для решения задач с различными характеристиками. Попробуем указать лишь некоторые широко распространенные задачи.

- Научно-технические расчеты, ориентированные на численное программирование.
- Экономические расчеты, связанные с доступом к базам данных и генерацией отчетов.
- Системы реального времени, требующие контроля планирования выполнения.
- Обработка текстов, манипуляции над строками символов.
- Графические пользовательские интерфейсы.

Этот выигрыш можно оценивать, в первую очередь, по общим затратам на разработку программного обеспечения. В этом контексте следует подчеркнуть влияние масштаба решаемых задач на выбор методологии программирования. Следующий немаловажный фактор — эффективность работы созданного программного обеспечения на современных компьютерах. И обязательно отметим, что некоторые методологии могут быть просто противопоказаны определенным задачам.

2.1.2. История и эволюция

Понять Бога легко, если только не пытаться его объяснить.
Жозеф Жубер

Достоверная информация о происхождении методологий может быть полезной для их более точной классификации и предсказания появления и развития новых методологий.

2.1.2.1. Методологии как концентрация практического опыта

Идея, примененная однажды, порождает искусственный прием.
Примененная дважды, она становится методом.
Г. Полиа, Г. Сеге

Первый и самый очевидный ответ на вопрос о происхождении методологий заключается в их независимом происхождении на основе практического опыта программистов. Заметим, что очень многие методологии имеют автора или основателя научной школы, исследующей данную методологию.

2.1.2.2. Алгоритмическое происхождение некоторых методологий

Это объяснение происходит из утверждения: "теория алгоритмов и логика — родители программирования". Четыре главные модели алгоритма, рассмотренные ниже, математически эквивалентны, но на практике они породили разные направления в программировании, в том числе некоторые основные методологии [Логика 1990].

- Первая модель — абстрактная вычислительная машина Тьюринга. Она определяет методологию императивного программирования.
- Вторая модель — рекурсивные функции Гильберта и Аккермана. От них унаследовала свои идеи и конструкции методология структурного программирования.
- Третья модель — лямбда-исчисление Черча, Шейнфинкеля и Карри. Эти идеи активно развиваются в методологии функционального программирования.
- Четвертая модель — нормальные алгорифмы Маркова. Некоторые специалисты считают, что эта модель послужила основой логического программирования.

2.1.2.3. Методологии как результат отображений структур

Еще одно объяснение берет за основу понятие отображения структур языка [Тузов 1990]. Согласно ему, сущность языка определяют три его составные части.

- Структура данных.
- Структура управления.
- Логика.

Каждая из трех структур языка моделирования может быть отображена на любую из структур языка программирования. В результате получается девять типов отображения. При этом каждое из отображений определяет либо методологию, либо, по крайней мере, достаточно серьезный метод.

1. Данные -> Данные. Отображение представляет собой процесс укрупнения данных и операций над ними и приводит к методам модульности и абстрактных типов данных.

2. Управление -> Управление. Отображение, связанное с понижением уровня структуры управления языка моделирования, ведет к идее методологии структурного программирования.
3. Логика -> Логика. Отображение лежит в основе методологии логического программирования.
4. Данные -> Управление. Отображение активизирует пассивные данные, преобразуя их в активные процессы, и лежит в основе методологии функционального программирования. В значительной степени это отображение определяет методологию объектно-ориентированного программирования.
5. Данные -> Логика. Отображение дает возможность по совокупности операций построить логическую структуру и определяет методологию программирования в ограничениях.
6. Управление -> Данные. Отображение лежит в основе методов интерпретации и определяет методологию доступ-ориентированного программирования.
7. Управление -> Логика. Отображение лежит в основе методов расшифровки смысла задачи.
8. Логика -> Данные. Отображение может быть связано с типизацией данных и определяет метод развитой системы типов и приведений. Также отображение может быть связано с интерпретаторами, реализующими языки с развитой логической структурой.
9. Логика -> Управление. Отображение может быть использовано в системах структурного синтеза.

Каждая из трех структур языка состоит из разнородных подструктур. Можно построить более точную классификацию, взяв за основу отображения этих подструктур.

2.1.3. Классификации методологий

2.1.3.1. Классификация по ядрам методологий

> Хотя все племена нтернети исповедуют, в общем, единый культ, между ними идут постоянные религиозные войны. Одна из самых затяжных связана с тем, какие палочки использовать для добывания дыма. Одно племя пользуется мсайе, палочками из древесины кокосовой пальмы, а другое — нтскейпами, из древесины банановой пальмы. Что интересно — те и другие одинаково кривые, однако каждое племя замечает лишь кривизну палочек противника.
>
> *Юрий Нестеренко*

Наш подход к методологиям заключается в том, что существует некоторое ядро методологии со своими методами, которое уточняется некоторыми дополнительными особенностями. Этот подход напоминает принцип словообразования в русском языке — есть корень, к которому добавляются приставки, суффиксы и окончания, уточняющие смысл слова.

Ядра методологий определяются способом описания алгоритмов. Перечислим основные ядра методологии и посвятим следующие разделы их анализу.

- Методология императивного программирования *(см. разд. 2.2.1)*.
- Методология объектно-ориентированного программирования *(см. разд. 2.2.2)*.
- Методология функционального программирования *(см. разд. 2.2.3)*.
- Методология логического программирования *(см. разд. 2.2.4)*.
- Методология программирования в ограничениях *(см. разд. 2.2.5)*.

Каково взаимоотношение между этими методологиями? Начнем с того, что первичной семантикой программирования является операционная семантика по очень простой причине. Программа бессмысленна без наличия вычислителя (машины), способного ее исполнить, причем исполнение протекает с достаточно мелкими шагами. Все остальные семантики, в конечном счете, основаны на операционной. Точно так же все основные разновидности программирования родились в недрах императивного и в большей или меньшей степени основаны на нем. И перечисленные методологии, кроме императивной, представляют собой ее развитие и, естественно, во многом обогащают.

Обратим внимание на то, что методологии находятся в диапазоне от навигационных, определяющих пошаговый контроль управления, до спецификационных, определяющих только требования к результату, но не задающих способ его получения.

2.1.3.2. Классификация по топологической специфике методологий

Топологическая специфика (топология) методологий определяется как способ выбора методов для получения уточненного ядра методологии. Каждое из "корней"-ядер может получить "приставку", определяемую некоторой топологией, которая может быть "хорошей" или "плохой". Что следует положить в основу критерия качества топологий? С нашей точки зрения, таким критерием может быть количество общих затрат на разработку программного обеспечения.

Затраты определяются совокупностью многочисленных факторов, в том числе связанных с абстракциями данных, управления и модульности *(см. разд. 4.3)*. Например, к хорошей топологии приводит отказ от использования глобальных данных и оператора безусловного перехода (за исключением особого ряда случаев), сильная связность модулей и их слабое сцепление.

Приведем пример. Если в императивной методологии придерживаться методов структурного программирования (дающих хорошую топологию с позиций всех упомянутых абстракций), то мы получим хорошо известную методологию структурного императивного программирования *(см. разд. 2.3.1)*,

которая более известна под ее кратким именем — методология структурного программирования. Вспомним, что одной из главных причин перехода к структурному программированию в начале 1970-х была необходимость снижения затрат на тестирование.

Обратим внимание на то, что успех объектно-ориентированной методологии изначально определила ее хорошая топология, базирующаяся на абстрактных типах данных.

2.1.3.3. Классификация по реализационной специфике методологий

Каждое из "корней"-ядер может получить "суффикс", определяющий некоторую организацию аппаратной поддержки данной методологии *(см. гл. 6)*. На данный момент наиболее известными организациями являются две: централизованная и параллельная.

Ядра методологий изначально продумывались для централизованных архитектур. Позже появились параллельные аппаратные реализации, к которым стали адаптировать уже существовавшие методологии. Приведем примеры параллельных методологий.

- Методология императивного параллельного программирования *(см. разд. 2.4.1)*. Эту методологию обычно называют кратко — методология параллельного программирования.
- Методология логического параллельного программирования *(см. разд. 2.4.2)*.

2.1.3.4. Смешанные методологии

Смешанные методологии включают объединение методов нескольких методологий. Наиболее часто объединяются методологии функционального и логического программирования. Существуют исследовательские работы в области объединения объектно-ориентированного и логического программирования (**http://www.math.spbu.ru/user/tseytin/papers.html**). Ряд исследований посвящен вопросам унификации методологий программирования [Hoare, Jifeng 1998].

2.2. Ядра методологий

2.2.1. Методология императивного программирования

Методология императивного программирования — подход, характеризующийся принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

2.2.1.1. Происхождение, история и эволюция

Императивное программирование — это исторически первая поддерживаемая аппаратно методология программирования. Она ориентирована на классическую фон Неймановскую модель *(см. разд. 6.2.1.1)*, остававшуюся долгое время единственной аппаратной архитектурой, получившей широкое практическое применение.

2.2.1.2. Методы и концепции

Метод *изменения состояний* заключается в последовательном изменении состояний. Метод поддерживается концепцией алгоритма *(см. разд. 1.7.3)*.

Метод *управления потоком исполнения* заключается в пошаговом контроле управления. Метод поддерживается концепцией *потока исполнения*.

2.2.1.3. Синтаксис и семантика

Основным синтаксическим понятием является оператор. Первая группа — атомарные операторы, у которых никакая их часть не является самостоятельным оператором (например, оператор присваивания, оператор безусловного перехода, вызова процедуры и т. п.). Вторая группа — структурные операторы, объединяющие другие операторы в новый, более крупный оператор (например, составной оператор, операторы выбора, цикла и т. п.).

Фрагмент синтаксических описаний для операторов в простейшем языке, поддерживающем императивную модель программирования, мог бы быть таким:

```
<оператор> ::= <простой оператор> | <структурный оператор>

<простой оператор> ::= <оператор присваивания> | <оператор вызова> |
<оператор возврата>

<структурный оператор> ::= <составной оператор> | <оператор ветвления> |
<оператор цикла>

<оператор присваивания> ::= <переменная> := <выражение>;

<оператор вызова> ::= <имя подпрограммы> ( [ <список параметров> ] );

<оператор возврата> ::= return [ <выражение> ];

<составной оператор> ::= begin <оператор> {; <оператор> } end

<оператор выбора> ::= if <выражение> then <оператор> | if <выражение>
then <оператор> else <оператор>

<оператор цикла> ::= while <выражение> do <оператор>
```

Императивное программирование основано на описании последовательного изменения состояний вычислителя. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех ячеек памяти, состояние процессора (в том числе указатель текущей команды) и всех сопряженных устройств. Единственная структура данных — последовательность ячеек (пар "адрес -> значение") с линейно упорядоченными адресами.

Языки, поддерживающие данную вычислительную модель, являются компактным средством описания функции переходов между состояниями вычислителя.

В качестве математической модели императивное программирование использует машину Тьюринга — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

Краткое описание семантики выглядит так. Операторы исполняются в порядке, предписанном объемлющем их структурным оператором. Если это составной оператор, то входящие в него операторы исполняются в том порядке, в котором записаны. Операторы, входящие в оператор выбора, могут исполняться или не исполняться в зависимости от значения логического условия. Исполнение атомарных операторов сводится к соответствующему единичному изменению состояния вычислителя. Исполнение вызова процедуры допускает возникновение по его ходу других вызовов.

Традиционное средство структурирования — подпрограмма (процедура или функция). Подпрограммы имеют параметры и локальные определения и могут быть вызваны рекурсивно. Функции возвращают значения как результат своей работы.

Если в данной методологии требуется решить некоторую задачу для того, чтобы использовать ее результаты при решении следующей задачи, то типичный подход будет таким. Сначала исполняется алгоритм, решающий первую задачу. Результаты его работы сохраняются в специальном месте памяти, которое известно следующему алгоритму, и используются им.

2.2.1.4. Языки императивного программирования

Языки императивного программирования манипулируют данными в пошаговом режиме, используя последовательные инструкции и применяя их к разнообразным данным. Считается, что первым алгоритмическим языком программирования был язык Plankalkuel (от plan calculus), разработанный в 1945—1946 годах Конрадом Цузе (Konrad Zuse).

Наиболее известные и распространенные императивные языки программирования, большинство из которых было создано в конце 50-х — середине 70-х годов XX века, представлены на рис. 2.1. Обратим внимание на пустое место на рисунке, соответствующее 80-м и 90-м годам прошлого века. Это период увлечения новыми парадигмами, и чистых императивных языков в это время практически не появлялось.

Программа на императивном языке ICON в листинге 2.1 демонстрирует типичный пример работы с текстом. Императивные языки легко и удобно использовать для написания экспериментальных и исследовательских приложений.

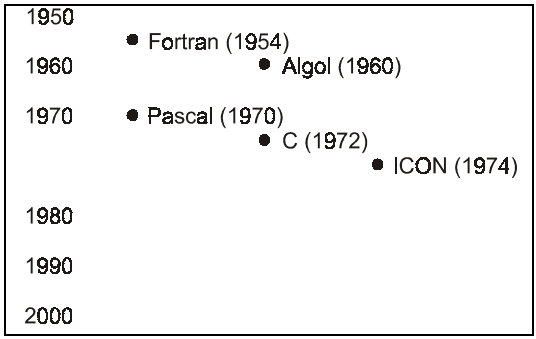


Рис. 2.1. Императивные языки программирования

Листинг 2.1. Пример программы на императивном языке

```
# Программа считывает слова, переданные параметром в командной строке,
# и печатает их в алфавитном порядке с указанием количества вхождений
procedure main(argv)
    T := table(0)         # создать таблицу
    every T[!argv] +:= 1 # каждый элемент командной строки используется
                          # как ключ в таблице для увеличения счетчика
    L := sort(T)          # сортировать таблицу, порождая
                          # список из ключа и значения
    while pair := pop(L) do           # брать очередной элемент из списка
        write(pair[1], ": ", pair[2]) # и печатать ключ и значение
end
```

2.2.1.5. Класс задач

Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Примером здесь может служить управление современными аппаратными средствами. Поскольку практически все современные компьютеры императивны, эта методология позволяет порождать достаточно эффективный исполняемый код.

С ростом сложности задачи императивные программы становятся все менее и менее читаемыми. Программирование и отладка действительно больших программ (например, компиляторов), написанных исключительно на основе методологии императивного программирования, может затянуться на долгие годы.

Вспомним, что первые задачи для компьютеров были рутинными вычислениями и представляли собой расчеты, связанные с созданием атомной бомбы.

2.2.2. Методология объектно-ориентированного программирования

Странный неопознанный объект летает над Стокгольмом.
Астрид Линдгрен. "Малыш и Карлсон"

Методология объектно-ориентированного программирования — это подход, использующий объектную декомпозицию, при которой статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

2.2.2.1. Происхождение, история и эволюция

На возникновение объектного мышления оказали влияние моделирование и представление данных, графические пользовательские интерфейсы и системное программирование (с понятием "процесс"). Исследования в области моделирования реальных систем привели к необходимости создания средств естественного описания сущностей, которые в них встречаются: объектов и событий. Позже оказалось, что такие концепции, как инкапсуляция (абстрактные типы данных), наследование и полиморфизм являются достаточно полезным дополнением к традиционному структурному программированию. Возможность их достаточно эффективной реализации привела к созданию широко распространенных в наши дни объектно-ориентированных языков.

2.2.2.2. Методы и концепции

Метод *объектно-ориентированной декомпозиции* заключается в выделении объектов и связей между ними. Метод поддерживается концепциями *инкапсуляции*, *наследования* и *полиморфизма*.

Метод *абстрактных типов данных* — это метод, лежащий в основе инкапсуляции. Метод поддерживается концепцией *абстрагирования (см. разд. 4.3.2.3)*.

Метод *пересылки сообщений* заключается в описании поведения системы в терминах обмена сообщениями между объектами. Метод поддерживается концепцией *сообщения*.

2.2.2.3. Синтаксис и семантика

Вычислительная модель чистого объектно-ориентированного программирования поддерживает явно только одну операцию, которой является посылка объекту сообщения. Сообщения могут иметь параметры, являющиеся объектами. Само сообщение также является объектом.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля — персональные переменные для данного объекта, значениями которых являются ссылки на другие объекты. В одном из полей объекта

хранится ссылка на объект-предок, которому переадресуются все сообщения, не обрабатываемые данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обычно выделяют в отдельный объект, называемый классом данного объекта. Сам объект называют экземпляром указанного класса.

Об объектно-ориентированных методах

Обратим внимание на то, что в этом разделе и далее до конца *разд. 2.2.2* термин "метод" заменяет традиционный термин "процедура" ("функция").

В семантике чистых объектно-ориентированных языков все может быть записано в форме посылки сообщений объектам. Класс в объектно-ориентированных языках описывает структуру и функционирование множества объектов с подобными характеристиками, атрибутами и поведением. Объект естественным образом принадлежит к некоторому классу и обладает своим собственным внутренним состоянием. Методы — это функциональные свойства, которые можно активизировать.

В объектно-ориентированном программировании определяют три основных свойства, перечисленные ниже.

- *Инкапсуляция* — это сокрытие информации и комбинирование данных и функций, которые аналогичны абстрактным типам данных. Инкапсуляция также имеет много общего с локализацией имен переменных и процедур в блоках, появившейся еще в языке Algol-60, и развивает ее.

О новом и старом

Итак, новое — это хорошо забытое старое. Абстрактный тип данных появился вновь через 25 лет после своего создания под названием объектно-ориентированный.

- *Наследование* — построение иерархии порожденных объектов с возможностью для каждого такого объекта, относящегося к иерархии, доступа к методам и данным всех порождающих объектов.

Об иерархиях

Любая иерархия — это компромисс. Построение иерархий является достаточно сложным делом. Объясняется это тем, что при этом приходится выполнять классифицирование *(см. разд. 1.7.1)*.

- *Полиморфизм* (полиморфизм включения) — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему. Таким образом, одно и то же имя дается разным действиям, способ исполнения которых задается в различных классах.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного

сообщения. Сообщение может быть передано вверх по иерархии наследования, если сам класс не располагает методом для его обработки. Если обработчик событий для сообщения выбирается динамически, то методы, реализующие обработчики событий, принято называть виртуальными.

В листинге 2.2 приведен пример описания класса "точка" в некотором абстрактном объектно-ориентированном языке.

Листинг 2.2. Пример программы на объектно-ориентированном языке

```
Object :subclass Point :fields { x, y };
Point :answer isnew :args { init_x, init_y } :body {
  Integer :new x :value { init_x };
  Integer :new y :value { init_y }
};
Point :answer getx :args {} :body { ^x };
Point :answer gety :args {} :body { ^y };
Point :answer setx :args {new_x} :body { x :set new_x };
Point :answer sety :args {new_y} :body { y :set new_y };
Point :answer move :args {delta_x, delta_y} :body {
  x :add delta_x
  y :add delta_y
};
// Если мы в дальнейшем хотим использовать класс Point, например,
// порождая его объекты и работая с ними, то код будет следующим:
Point :new A :value {0, 0};
A :move {+2, -2};
```

Естественным средством структурирования в данной методологии являются классы. Классы определяют, какие поля и методы экземпляра доступны извне, как обрабатывать отдельные сообщения и т. п. В чистых объектно-ориентированных языках извне доступны только методы.

Взаимодействие задач в данной методологии осуществляется при помощи обмена сообщениями между объектами, реализующими данные задачи.

Заметим, что объектно-ориентированная методология естественным образом близка к параллельной. Каждый отдельный объект становится процессом, находящимся в постоянном ожидании сигналов-сообщений. Кстати, одним из первых представлений об объектах является экторная модель параллельных вычислений Хьюитта. Под объектом в ней подразумевается

активный процесс, получающий сообщения и изменяющий свое внутреннее состояние в зависимости от полученного сообщения. Для этой модели справедливы следующие свойства:

- объектом является процесс, который может иметь различные внутренние состояния. При получении сообщения объект становится активным;
- извне внутреннее состояние объекта может быть изменено только посредством передачи ему сообщения, специфицирующего выполняемую объектом операцию;
- во время работы объект может обмениваться сообщениями с другими объектами.

2.2.2.4. Языки объектно-ориентированного программирования

Языки объектно-ориентированного программирования содержат конструкции, позволяющие определять объекты, принадлежащие классам и обладающие свойствами инкапсуляции, наследования и полиморфизма. Такие языки можно разделить на три группы (рис. 2.2).

- *Чистые* языки, в наиболее классическом виде поддерживающие объектно-ориентированную методологию. Такие языки обычно содержат существенную библиотеку, а также набор средств поддержки времени исполнения.
- *Гибридные* языки, которые появились в результате внедрения объектно-ориентированных конструкций в популярный императивный язык программирования.
- *Урезанные* (*очищенные*) языки, которые появились в результате удаления из гибридных языков наиболее опасных и ненужных с объектно-ориентированной точки зрения конструкций.

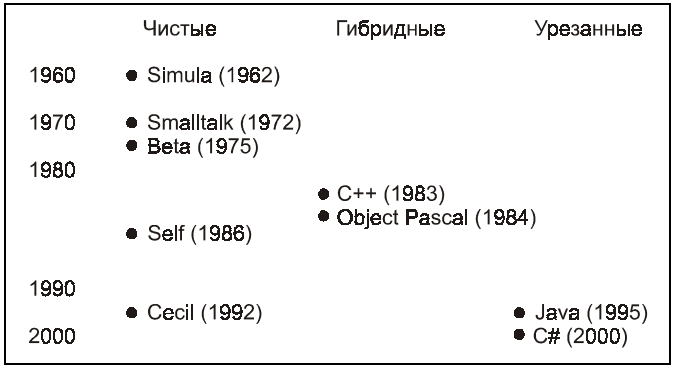


Рис. 2.2. Объектно-ориентированные языки программирования

2.2.2.5. Класс задач

Данная методология является мощным средством для моделирования отношений между объектами практически в любой предметной области. Особенно удобно и легко в объектах выразить взаимодействие между различными элементами графического интерфейса пользователя. Отметим, что категорически не рекомендуется применять эту методологию для систем реального времени с жесткими ограничениями на время.

2.2.3. Методология функционального программирования

Нужно быть терпеливым, чтобы научиться терпению.
Станислав Ежи Лец

Методология функционального программирования — способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части — введение имени для функции и задание для этого имени выражения, вычисляющего значения функции, а единственным правилом композиции — оператор суперпозиции функции.

2.2.3.1. Происхождение, история и эволюция

Функциональная методология является одной из старейших. По происхождению она тесно связана с лямбда-исчислением, изобретенным еще в начале 30-х годов XX века логиком Алонзо Черчем (Alonzo Church). Для многих функциональная методология стала ассоциироваться с языком Lisp, созданным Джоном Маккарти (John McCarthy) в конце 50-х годов XX века [Хювенен, Сеппянен 1986]. В то же время эта методология в основном используется теоретиками программирования и является средством лабораторных исследований искусственного интеллекта.

2.2.3.2. Методы и концепции

Метод *аппликативности* заключается в том, что программа есть выражение, составленное из применения функций к аргументам. Программа состоит из совокупности определений функций, представляющих собой вызовы других функций и вложенных друг в друга. Метод поддерживается концепцией *функции*.

Метод *рекурсивного поведения* заключается в самоповторяющемся поведении, возвращающемся к самому себе. Метод поддерживается концепцией *рекурсии*.

Метод *настраиваемости* заключается в том, что можно легко порождать новые программные объекты по образцу, как значения соответствующих выражений (применение порождающей функции к параметрам образца). Этому

способствует то, что не только программа, но и любой программный объект (в идеале) является выражением.

2.2.3.3. Синтаксис и семантика

Функциональное программирование представляет собой одну из альтернатив императивному подходу. В функциональном программировании отсутствует понятие времени. Программы являются выражениями, а исполнение программ заключается в вычислении этих выражений.

При описании функционального программирования, как правило, рассматривают так называемое "расширенное лямбда-исчисление".

Его грамматика может выглядеть следующим образом:

```
<выражение> ::= <простое выражение> | <составное выражение>

<простое выражение> ::= <константа> | <имя>

<составное выражение> ::= <лямбда-абстракция> | <применение> |
<квалифицированное выражение> | <ветвление>

<лямбда-абстракция> ::=  lambda <имя> → <выражение> end

<применение> ::= (<выражение> <выражение>)

<квалифицированное выражение> ::= let <имя> = <выражение> {; <имя> =
<выражение>} in <выражение> end

<ветвление> ::= if <выражение> then <выражение> end | if <выражение> then
<выражение> else <выражение> end
```

Константами в расширенном лямбда-исчислении могут быть числа, кортежи, списки, имена предопределенных функций и т. п. Результатом вычисления применения предопределенной функции к аргументам будет значение предопределенной функции в этой "точке". Результатом применения лямбда-абстракции к аргументу будет подстановка аргумента в выражение — "тело" лямбда-абстракции. Сами лямбда-абстракции также являются выражениями и, следовательно, могут быть аргументами.

Обратим внимание на то, что лямбда-абстракции имеют всего один аргумент. В то же время, функции в традиционном понимании не обязаны быть одноместными. Представления функций от нескольких аргументов можно достичь двумя способами.

- В первом способе будем считать, что аргумент является кортежем. Например,

  ```
  apply = lambda (f, x) → (f x) end
  ```

  можно понимать как

  ```
  apply = lambda y → ((first y) (second y)) end.
  ```

- Второй способ исходит из того, что множество вычислимых функций `X * Y → Z` очевидным образом взаимно однозначно отображается в множество вычислимых функций `X → (Y → Z)`. Так,

```
apply = lambda f → lambda x → (f x) end end.
```

О расстановке скобок

Вместо расстановки скобок вокруг применения функций к аргументам, можно объявить операцию применения функции (которую мы при записи опускаем так же, как в математике принято не писать явно символ умножения) левоассоциативной. Будем понимать запись вида `f x y` как `((f x) y)`.

Чистое лямбда-исчисление Черча позволяет обходиться исключительно именами параметров, лямбда-абстракциями от одного аргумента и применениями выражений к выражениям. В этих терминах можно описать и "предопределенные" константы (например, числа), структуры данных (например, списки и кортежи), логические значения и ветвление. Более того, в чистом лямбда-исчислении можно обойтись без квалифицированных выражений и, следовательно, выразить рекурсию, не используя для этого употребления имени функции в теле функции. Некоторые экспериментальные модели функционального программирования позволяют обходиться без каких-либо обозначений объектов (имен) вообще.

Для обеспечения видовой корректности программ в функциональные языки вводят специальные системы типов, ориентированные на поддержку настраиваемости. Как правило, трансляторы функциональных языков могут самостоятельно определять типы выражений без каких-либо описаний типов вообще. Так, функция

```
add = lambda x → lambda y → x+y end end
```

будет иметь тип `number → number → number`, а уже рассматриваемая нами функция `apply` — тип `any(X).any(Y).(X→Y) →X→Y`, где `any` обозначает "квантор всеобщности" для типов, а `X` и `Y` являются переменными.

Нередко существует возможность в качестве "списка параметров" для функции указать "образец", успешное сопоставление с которым определяет выражение, которое будет вычисляться (листинг 2.3).

Листинг 2.3. Пример программы на функциональном языке

```
let
    map = fun
        (f, [ ]) → [ ];
        (f, [head|tail]) → [f(head)|map(f,tail)]
        end fun;
```

```
    square = fun
        x → multiply(x, x)
        end fun
in
    map(square, [1,2,3])
end let
```

Так как порядок вычисления подвыражений не имеет значения (поскольку "состояния" у функциональной программы нет), функциональное программирование может быть естественным образом реализовано на платформах, поддерживающих параллелизм. "Потоковая модель" функционального программирования [Филд, Харрисон 1993] является естественным представлением функциональных программ в терминах систем взаимодействующих процессов.

2.2.3.4. Языки функционального программирования

Основная специфика языков функционального программирования заключается в том, что функции обмениваются между собой данными непосредственно, т. е. без использования промежуточных переменных и присваиваний. Переменные, однажды получив значение, никогда его не изменят. Присваивание (например, в императивной методологии) — это конструкция, вносящая время в процесс вычисления. Правая часть присваивания должна быть вычислена и только после этого ее значение будет связано с левой частью. В результате, в функциональных языках циклы заменяются аппаратом рекурсивных функций. Благодаря отсутствию присваивания, программы на функциональных языках вообще не содержат так называемых побочных эффектов.

На рис. 2.3 представлены основные функциональные языки. Первый пик интереса к ним приходится на конец 70-х — начало 80-х годов XX века. Второй пик приходится на настоящее время, и он связан в первую очередь со стандартизацией наиболее мощного чистого функционального языка — Haskell.

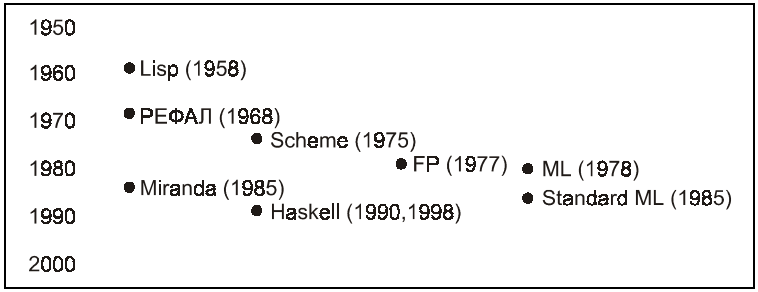


Рис. 2.3. Функциональные языки программирования

Особенность ранних функциональных языков — в их динамической типизации. Типы формируются вместе с их значениями в ходе вычислений. За последние 15 лет функциональное программирование перестало быть исключительно исследовательским. Современные языки функционального программирования (ML, Haskell) строго типизированные, компилирующиеся, предназначенные как для индивидуальной работы над программами, так и для коллективной разработки.

2.2.3.5. Класс задач

Функциональное программирование обычно применяется для решения тех задач, которые трудно сформулировать в терминах последовательных операций. В эту категорию попадают практически все задачи, связанные с искусственным интеллектом. Это такие задачи, как обработка естественного языка, экспертные консультирующие системы, проблемы зрительного восприятия и многие другие.

2.2.4. Методология логического программирования

Я мыслю, следовательно, я существую.
Декарт (и Роденовский мыслитель)

Я не мыслю, следовательно, я не существую.
Г. Лихтенберг

Методология логического программирования — подход, согласно которому программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система выполняет с помощью механизмов логического вывода.

2.2.4.1. Происхождение, история и эволюция

Логическое программирование начинает свой отсчет времени с конца 60-х годов XX века, когда Корделл Грин (Cordell Green) предложил использовать резолюцию как основу логического программирования. Алан Колмероэ (Alain Colmerauer) создал язык логического программирования Prolog в 1971 году. В основе логических языков лежит теория хорновских дизъюнктов. Логическое программирование пережило пик популярности в середине 80-х годов XX века, когда оно было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения [Язык 1988].

2.2.4.2. Методы и концепции

Метод *единообразия* заключается в одинаковом применении механизма логического доказательства ко всей программе.

Метод *унификации* — это механизм сопоставления с образцом для создания и декомпозиции структур данных.

2.2.4.3. Синтаксис и семантика

В логике теории задаются при помощи аксиом и правил вывода. То же самое мы имеем и в базисном языке логического программирования Prolog. Аксиомы здесь принято называть фактами, а правила вывода ограничить по форме до так называемых "дизъюнктов Хорна" — утверждений вида `A <= B1& ...& Bn`. В языке Prolog такие утверждения принято записывать следующим образом:

```
a :- b1, ..., bn.
```

Факты (они же аксиомы) представляются как правила с пустой "посылкой":

```
a.
```

Переменные в утверждениях принято обозначать идентификаторами, начинающимися с заглавной буквы. Пример простейшей программы на языке Prolog выглядит так:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Эту программу можно прочитать так: "`X` является членом списка, если он совпадает с головой списка, или является членом хвоста списка". В этой программе объявлен один предикат — `member`.

Обычно Prolog-система работает в форме диалога с пользователем. Утверждение, которое требуется доказать, вводится с клавиатуры. Компилирующие версии трансляторов Prolog могут располагать специальными синтаксическими средствами для задания утверждений, которые требуется доказать. Такие утверждения в Prolog принято называть целями.

Зададим системе простейший вопрос: "Является ли 2 членом списка [1, 2, 3]?" Для этого введем:

```
?- member(2, [1,2,3]).
```

Prolog-система сначала попытается применить первое правило для предиката `member`, сравнивая `2` с головой списка `[1,2,3]`. Это сравнение не является успешным. Система продолжит вывод по второму правилу, рекурсивно вызывающему предикат `member`, с аргументами `2` и `[2,3]`. В этом рекурсивном вызове можно применить первое правило (т. к. `2` совпадает с головой списка `[2,3]`), и тогда Prolog-система выдаст ответ:

```
yes ->
```

Так как произвольная цель может быть доказана не единственным образом, система предлагает нам решить, пытаться ли доказать это утверждение по-другому. Ответим "да" тем способом, как это предусмотрено в используемой Prolog-системе. Осталось незадействованным второе правило для предиката `member` для аргументов `2`, `[2,3]`, по которому следует пытаться доказать, что `2` есть член списка `[3]`. Так как `2 != 3`, первое правило для этой цели не сработает, и доказательство пойдет дальше по второму правилу, которое

предписывает доказывать утверждение `member(2, [])`. Так как у пустых списков нет головы и хвоста, ни одно из правил для предиката `member` не применимо, и Prolog-система выдаст ответ:

```
no.
```

Мы могли бы задать Prolog-системе и более "интересные" вопросы, например "для какого `X` верно, что он является членом списка `[1,2,3]`?":

```
?- member(X, [1,2,3]).
```

Система последовательно выдаст нам варианты:

```
X=1 ->
X=2 ->
X=3 ->
no.
```

Можно задать и еще один "интересный" вопрос: "для какого `X` верно, что `1` является членом списка `X`?":

```
?- member(1, X).
X = [1|_] ->
X = [_, 1|_] ->
X = [_, _, 1|_] ->
...
```

Таких списков, очевидно, бесконечно много. Рано или поздно нам надоест наблюдать порождаемые Prolog-системой списки, содержащие `1`, и мы скажем "нет" на очередной запрос о продолжении доказательства. Система выдаст нам долгожданное "`no.`" и закончит доказательство утверждения `member(1, X)`.

Prolog-система использует метод унификации и метод резолюций для доказательства утверждений. *Унификация* — это сопоставление двух произвольных термов, содержащих переменные, с целью определения того, можно ли присвоить этим переменным такие значения, чтобы получились два одинаковых терма. Например, унификация термов `f(X, 2)` и `f(1, Y)`, где `X`, `Y` — переменные, выдаст подстановку: `X=1`, `Y=2`. Унификация термов `f(X)` и `X` пройдет безуспешно. *Метод резолюций* заключается в последовательном доказательстве отдельных утверждений, входящих в посылку дизъюнкта Хорна, для доказательства его следствия. То есть применение метода резолюций к правилу `a :- b, c.` и утверждению `a` приведет к последовательному доказательству утверждений `b` и `c`. Метод резолюций имеет прямой аналог в обычной логике высказываний — правило modus ponens, по которому `(A & (A=>B)) => B`.

2.2.4.4. Языки логического программирования

Языки логического программирования содержат конструкции, позволяющие выполнить описание проблемы в терминах фактов и логических формул, а собственно решение проблемы выполняет система с помощью механизмов

логического вывода. От языка программирования неотъемлем механизм конструктивного вывода целевого утверждения, основанный на строгих математических моделях. Язык Prolog является родоначальником семейства ему подобных языков (рис. 2.4), в котором можно выделить три ветви.

- Модификации языка (использование более мощных логических средств, внесение модульности и т. п.).
- Функциональное направление (комбинация с функциональными языками).
- Параллельное направление (логическое программирование по сути своей параллельно).

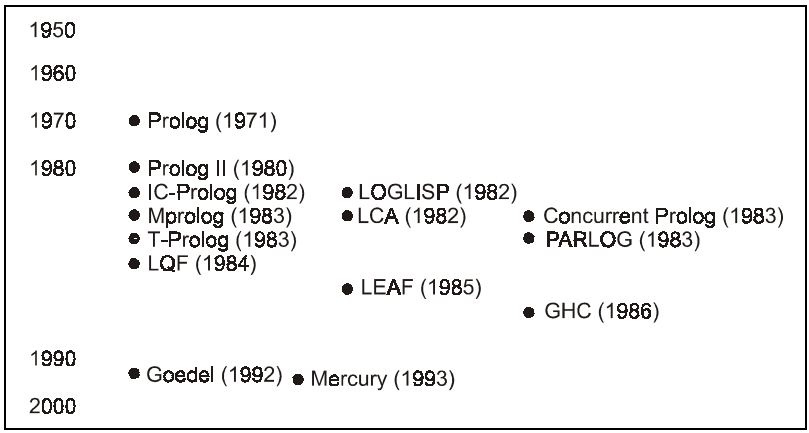


Рис. 2.4. Логические языки программирования

2.2.4.5. Класс задач

Класс задач логического программирования практически совпадает с классом задач функционального программирования.

2.2.5. Методология программирования в ограничениях

Чем дальше мы продвигались по Южным штатам, тем чаще сталкивались со всякого рода ограничениями, устроенными для негров.

Илья Ильф, Евгений Петров. "Одноэтажная Америка"

Методология программирования в ограничениях — это подход, при котором в программе определяется тип данных решения, предметная область решения и ограничения на значение искомого решения. Решение находится системой. Методология предлагает двухуровневую архитектуру, интегрирующую компонент ограничения и программный компонент. Компонент ограничений обеспечивает основные операции и состоит из системы выводов на фундаментальных свойствах системы ограничений. Операции, окружающие компонент ограничений, реализуются программно-языковым компонентом.

2.2.5.1. Происхождение, история и эволюция

Методология возникла в начале 80-х годов XX века как перспективная область исследований на стыке символьных вычислений, искусственного интеллекта, исследования операций и интервальной арифметики.

2.2.5.2. Методы и концепции

Метод *описательной модели вычислений* заключается в том, что программа на языке программирования содержит описание понятий и задач. Метод поддерживается концепцией *модели*.

2.2.5.3. Синтаксис и семантика

Программирование в ограничениях — это программирование в терминах постановок задач.

Постановка задачи — это конечный набор переменных `V = {v[1], ..., v[n]}`, соответствующих им конечных (перечислимых) множеств значений `D = {D[1], ..., D[n]}`, и набор ограничений `C = {C[1], ..., C[m]}`. Ограничения представлены как утверждения, в которые в качестве "параметров" входят переменные из некоторого подмножества `V[j], j=1..m` набора `V`. Решение такой задачи — набор значений переменных, удовлетворяющий всем ограничениям `C[j]`.

Семантически исполнение программы рассматривается как нахождение значений переменных. При этом порядок удовлетворения отдельных ограничений не имеет значения, и система обычно стремится оптимизировать порядок нахождения значений с целью минимизировать откат в случае неуспеха.

С этой целью набор ограничений может быть соответствующим образом преобразован по правилам, аналогичным правилам языка Prolog. Любую задачу можно рассматривать как ограничение: "значения переменных должны быть решением этой задачи". В качестве примера, мы можем задать системе программирования в ограничениях следующий запрос:

```
? (X : integer) X>1, member(X, [1,2,3]).
```

Обратим внимание на то, что типичная Prolog-система на таком запросе выдаст ошибку. `X` является неинициализированной переменной, и его нельзя сравнивать с числом `1`. Система, поддерживающая программирование в ограничениях, воспримет эти утверждения как ограничения, а не как цели, которые требуется доказать. В результате будут выданы требуемые решения: `X=2` и `X=3`.

Задачу в ограничениях можно рассматривать и как задачу о минимально необходимом наборе ограничений, что позволяет в некоторых случаях описать в явном виде бесконечное множество решений. Так, минимально необходимым

набором ограничений для задачи `X:integer, X>2, X<4` будет `X=3`. Для задачи `X,Y:integer, X*2=Y` таким набором будет `X:integer, Y mod 2 = 0`.

Системы символьных вычислений нередко позволяют использовать "допущения", по сути являющимися ограничениями. На следующий простой запрос:

```
assume X>0.
when X+1<10 ?
```

системы могут выдать такой ответ:

```
X in (1..8).
```

Как правило, такие системы могут доказывать достаточно нетривиальные математические утверждения.

2.2.5.4. Языки программирования в ограничениях

Языки программирования в ограничениях позволяют в программе определить тип данных решения, предметную область решения и ограничения на значение искомого решения. Решение находится системой. Данные языки получили наибольшую известность в 80-х годах XX века (рис. 2.5).

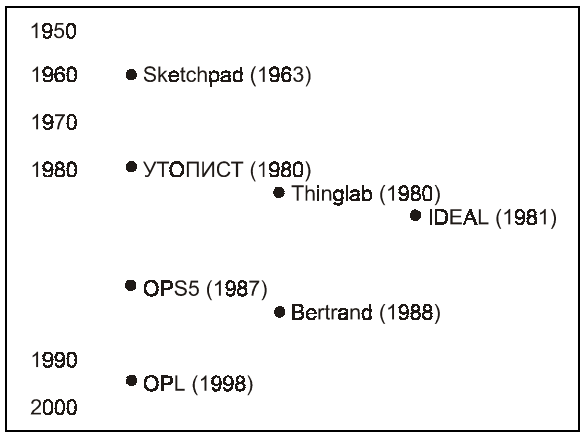


Рис. 2.5. Языки программирования в ограничениях

Заметим, что зачаточным языком программирования в ограничениях может служить язык описания условий для утилиты make.

Язык программирования УТОПИСТ (Универсальные Текстовые ОПИСания Терминов) предназначен для описания понятий и задач. Язык является базовым для инструментальной системы программирования ПРИЗ [Кахро, Калья, Тыугу 1988]. Язык имеет процедурную часть (поскольку при описании задачи иногда приходится описывать действия), но основная его выразительность достигается за счет описаний.

Приведем пример модели и постановки задачи на языке УТОПИСТ для вычисления некоторых значений на основе закона Ома (листинг 2.4).

Листинг 2.4. Пример программы на языке программирования в ограничениях

```
ПУСТЬ'
    СХЕМА:(I1, I2, I3, U1, U2, U3, R1, R2, R3: ВЕЩ';
       УРАВ' U1 = I1 * R1;
       УРАВ' U2 = I2 * R2;
       УРАВ' U3 = I3 * R3;
       УРАВ' I1 + I2 + I3 = 0);
    СХЕМА1: СХЕМА
       R1 = 16, R2 = 32, R3 = 5,
       U1 = 50, U2 = -28;
ДЕЙСТВИЯ'
    НА' СХЕМА1 ВЫЧИСЛИТЬ' U3;
```

2.2.5.5. Класс задач

Класс задач программирования в ограничениях — это задачи исследования операций и искусственного интеллекта. В таких задачах часто используется некоторое пространство решений, сужением которого достигается необходимый результат. Такие сужения можно естественным образом представить как ограничения.

2.3. Топологическая специфика методологий

2.3.1. Методология структурного императивного программирования

Методология структурного императивного программирования — подход, заключающийся в задании хорошей топологии императивных программ, ориентированной на сокращение количества общих затрат на разработку программного обеспечения. Сокращение будет иметь место в результате того, что и проектные модели, и программный код будут иметь хорошую структурированность, позволяющую избежать многих ошибок. В случае данной методологии хорошую топологию задают отказ от использования глобальных данных и, в большинстве случаев, оператора безусловного перехода, разработка модулей с сильной связностью и обеспечение их

независимости от других модулей. Подход базируется на двух основных принципах построения:

- последовательная декомпозиция алгоритма решения задачи сверху вниз;
- использование структурного кодирования.

Напомним, что данная методология является важнейшим развитием императивной методологии.

2.3.1.1. Происхождение, история и эволюция

Создателем структурного подхода считается Эдсгер Дейкстра. Ему также принадлежит попытка (к сожалению, в настоящее время совершенно неприменимая для массового программирования) соединить структурное программирование с методами доказательства программ [Дейкстра 1978].

2.3.1.2. Методы и концепции

Метод *алгоритмической декомпозиции сверху вниз* заключается в пошаговой детализации постановки задачи, начиная с наиболее общей задачи. Данный метод обеспечивает хорошую структурированность и поддерживается концепцией *алгоритма (см. разд. 1.7.3)*.

Метод *модульной организации частей программы* заключается в разбиении программы на специальные компоненты, называемые модулями. Метод поддерживается концепцией *модуля (см. разд. 4.3.4)*.

Метод *структурного кодирования* заключается в использовании при кодировании трех основных управляющих конструкций. Метки и оператор безусловного перехода являются трудно отслеживаемыми связями, без которых мы хотим обойтись. Метод поддерживается концепцией *управления (см. разд. 4.3.3)*.

2.3.1.3. Языки структурного программирования

Основное отличие от классической методологии императивного программирования заключается в структурности программ и в трех способах композиции операторов: последовательности двух или более операторов, дихотомического выбора и повторения.

Если мы возьмем конструкцию повторения (оператор цикла) `while B do S end`, то она имеет важное свойство — после выполнения оператора условие `B` будет иметь значение "ложь" независимо от оператора `S`. Если же находится свойство, которое является инвариантом (т. е. остается истинным, несмотря на выполнение части программы) по отношению к `S`, то оно останется таким же по завершении. Такие конструкции помогают в надежном создании программ и сокращают время, которое тратится на тестирование и отладку.

Немаловажной особенностью будет отказ (точнее, та или иная степень отказа) от *оператора безусловного перехода.*

Практически на всех языках, поддерживающих императивную методологию, можно разрабатывать программы и по данной методологии. Четыре основных мнения по поводу оператора безусловного перехода, поддерживаемого различными языками программирования, приведены в табл. 2.1.

Таблица 2.1. *Четыре основных мнения по поводу оператора* `goto`

Точка зрения	Заменители `goto`	Использование `goto`	Примеры языков
Ортодоксальная (Дейкстра)	Нет	Нет	BLISS, PDL
Только заменители `goto`	Да	Нет	Java, Modula-2
Критики Дейкстры (только `goto`)	Нет	Да	BASIC
И заменители, и `goto`	Да	Да	C, Ada

2.3.1.4. Класс задач

Класс задач для данной методологии соответствует классу задач для императивной методологии. Заметим, что при этом удается разрабатывать более сложные программы, поскольку их легче воспринимать и анализировать.

2.4. Реализационная специфика методологий

2.4.1. Методология императивного параллельного программирования

Методология императивного параллельного программирования — подход, в котором предлагается использование явных конструкций для параллельного исполнения выбранных фрагментов программ.

2.4.1.1. Происхождение, история и эволюция

Вычислительные задачи часто имеют огромные объемы. Анализ эффективности их решений показал, что ситуацию можно значительно улучшить, если использовать для вычислений не одно, а несколько вычислительных устройств одновременно. Создание аппаратных многопроцессорных архитектур привело к широким исследовательским работам в этой области. Еще одна причина возникновения данной методологии связана с появлением достаточно

сложных программ, требующих поддержки явного параллелизма (например, операционных систем).

Считается, что параллельное программирование возникло в 1962 году с изобретением каналов — независимых аппаратных контроллеров, позволявших центральному процессору выполнять новую прикладную программу одновременно с операциями ввода-вывода других программ. Первоначально с параллельным программированием имели дело лишь разработчики операционных систем.

2.4.1.2. Методы и концепции

Метод *синхронизации* исполняемого кода заключается в использовании специальных атомических операций для осуществления взаимодействия между одновременно исполняемыми фрагментами кода. Метод поддерживается концепцией *примитивов синхронизации (см. разд. 7.2.2.1)*.

2.4.1.3. Синтаксис и семантика

Модель вычислений параллельного программирования рассмотрим в контексте модели императивного программирования. Имеется несколько вычислителей, которые характеризуются наличием общих элементов состояния. Одно из возможных описаний вычислительной модели выглядит следующим образом. Если на шаге вычислений с некоторым номером элемент состояния одного вычислителя принимал данное значение, то и для любого другого вычислителя на шаге вычисления с тем же номером соответствующий элемент состояния также должен принимать данное значение.

Прямым аналогом оператора в данной методологии является процесс. Основное отличие этой методологии от императивной в том, что процессы могут исполняться параллельно.

Параллельная методология тесно привязана к платформе, на которой она реализуется. Параллелизм может быть реализован как аппаратно на централизованной архитектуре, так и в распределенной архитектуре. Как правило, всегда имеются команды типа "породить потомка", "дождаться завершения работы потомка" и т. п.

В общем случае параллелизм заключается в одновременной обработке приложений, процессов, подпрограмм, циклов и операторов. В зависимости от уровня дробления выделяют следующие уровни параллелизма:

- параллелизм на уровне микрокоманд;
- параллелизм на уровне операторов (кроме циклов);
- параллелизм на уровне циклов и итераций;

- параллелизм на уровне подпрограмм (процедур и функций);
- параллелизм на уровне потоков управления;
- параллелизм на уровне процессов;
- параллелизм на уровне приложений.

Традиционное средство структурирования высокого уровня в данной методологии — процесс. Процессы могут работать параллельно. Они осуществляют синхронизацию между собой с помощью примитивов синхронизации (например, с помощью каналов связи). Приведем модель взаимодействия процессов, достаточно близкую к реальным языкам параллельного программирования:

```
<процесс> ::= <простой процесс> | <структурный процесс>

<простой процесс> ::= <послать значение> | <принять значение> | <процесс
вычислительной модели>

<структурный процесс> ::= <последовательный процесс> | <параллельный
процесс>

<послать значение> ::= <канал связи> << <выражение>;

<принять значение> ::= <канал связи> >> <выражение>;

<процесс вычислительной модели> ::= <любые вычислительные действия, не
зависящие от наличия параллелизации>

<последовательный процесс> ::= seq <процесс> { <процесс> } end

<параллельный процесс> ::= par <процесс> { <процесс> } end
```

Семантически взаимодействие параллельных процессов лучше всего представлять как работу сети неких устройств, соединенных каналами, по которым текут данные. Спроектировав в таких терминах, например, систему параллельных процессов для быстрого суммирования большого количества чисел, можно легко описать эту систему в приведенном синтаксисе. Каждый вычислитель производит типичные для его вычислительной модели операции (например, императивный вычислитель будет переходить из состояния в состояние). Когда процесс встречает инструкцию "принять значение из канала", он входит в состояние ожидания, пока канал пуст. Как только в канале появляется значение, процесс его считывает и продолжает работу.

Суперпозиция задач, как правило, реализуется через систему процессов, обменивающихся между собой информацией о результатах своих вычислений.

2.4.1.4. Языки параллельного программирования

Языки параллельного программирования используют явные конструкции для параллельного исполнения выбранных фрагментов программ. Существует

несколько языковых подходов к программированию для параллельных вычислительных систем.

- Программирование на параллельном языке программирования. Причем такие языки могут быть:
  - универсальными (например, Ada);
  - для конкретных типов компьютеров, позволяющих эффективно транслировать программы на параллельном языке именно в эту архитектуру (например, язык Occam изначально разрабатывался для транспьютеров).
- Программирование на широко распространенном языке программирования (например, C, C++, Pascal), который расширен языковыми (на уровне языка программирования) распараллеливающими конструкциями.
- Программирование с использованием дополнительных указаний компилятору на уровне языка прагм (например, по стандарту OpenMP).
- Программирование на широко распространенном языке программирования с использованием коммуникационных библиотек и интерфейсов для организации межпроцессорного взаимодействия. В этом случае конструкции параллелизма вынесены с языкового уровня на уровень операционной системы.
- Применение средств автоматического распараллеливания последовательных программ такими инструментами, как компиляторы.

На рис. 2.6 приведены наиболее распространенные языки программирования, содержащие явные конструкции параллельного исполнения.

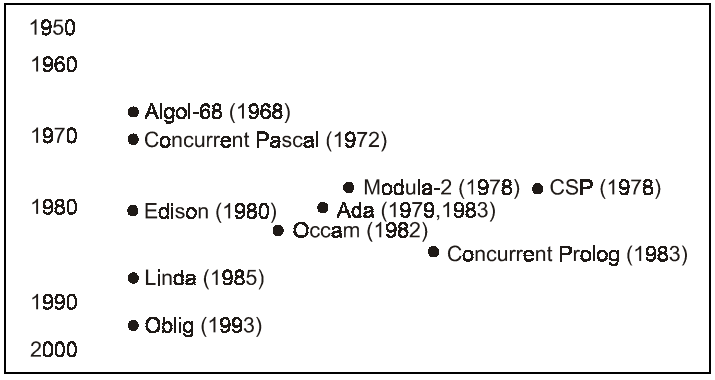


Рис. 2.6. Параллельные языки программирования

2.4.1.5. Класс задач

Данная методология может очень эффективно применяться для обработки больших однородных массивов данных. Такие массивы часто встречаются в

реализации вычислительных и статистических методов. Кроме этого, методология параллельного программирования успешно применяется при моделировании, в операционных системах и системах реального времени *(см. разд. 7.2)*.

2.4.2. Методология логического параллельного программирования

> В моих рассуждениях была все-таки некоторая логика, ускользавшая от Бертрана. Я сказала ему об этом, мы стали спорить. В заключение он привел меня к себе без обеда, в виде наказания.
>
> *Франсуаза Саган. "Смутная улыбка"*

Логическое программирование допускает естественную параллельную реализацию. В примере `a :- b, c.` порядок согласования целей `b` и `c` не имеет значения, поэтому их можно доказывать параллельно. Говорят, что процессы доказательства `b` и `c` образуют *И-систему* процессов. И-система успешно доказывается, если каждый процесс, входящий в систему, успешен. В примере с предикатом `member` *(см. разд. 2.2.4.3)* два правила для него могли применяться параллельно, образуя *ИЛИ-систему* процессов. ИЛИ-система успешно доказывается, если хотя бы один процесс в системе успешен. Переменные, общие для системы процессов (например, в случае `a(X) :- b(X), c(X).`), преобразуются в каналы связи между процессами в системе. Связывание переменной (присвоение ей значения) аналогично посылке значения в канал.

2.4.2.1. Языки логического параллельного программирования

Одним из наиболее ярких языков логического параллельного программирования является язык Concurrent Prolog, предложенный Эхудом Шапиро (Ehud Shapiro) [Язык 1988]. Язык развивает логический подход к абстрактным спецификациям и включает несколько идей Дейкстры (в том числе, идею охраняемого предложения). При этом подходе предметная область описывается как формальная теория в некотором логико-математическом языке. Когда мы используем абстрактную модель вычисления логических программ, то предполагаем, что они легко поддаются параллельному выполнению. Вычисление логической программы — это доказательство целевого утверждения с использованием аксиом из программы. Пространство поиска вывода может быть описано И/ИЛИ деревом, где И-узлы соответствуют конъюнкции целей, а ИЛИ-узлы соответствуют различным путям выбора единичных целей из программы.

В языке Concurrent Prolog применяется особое прочтение логических программ — поведенческое. В поведенческом прочтении единичная цель есть аналог процесса, конъюнкция целей — аналог системы процессов, а общие переменные целевых функций — просто каналы связи. Предложение читается поведенчески: процесс `A` может заменить себя системой процессов `B1`, `B2`, ... и `Bn`. Процесс оканчивается заменой себя пустой системой.

Язык Concurrent Prolog поддерживает различные средства объектно-ориентированного программирования. Однако они сильно зависят от свойств унификации. В объектно-ориентированных языках сообщение посылается с указанием имени целевого объекта. В Concurrent Prolog объекты связаны общими переменными, и сообщения посылаются установкой общих переменных, поэтому имя целевого объекта не обязательно должно появляться в фазе посылки сообщения. Более того, передача становится тривиальной, т. к. сообщение одновременно посылается всем объектам, которые разделяют переменную.

Как правило, общие переменные возникают, когда процесс создает новую систему процессов, как в следующем случае:

```
p(X) :- q(X, Y), r(Y?).
```

`Y` создается и используется как канал связи от `q` к `r`. Общие переменные могут создаваться динамически при посылке переменной как части сообщения. Это означает, что канал связи может быть создан динамически и отправлен другому объекту. Сообщение, содержащее переменную, называется незавершенным (частично определенным) сообщением. Оно делает новую переменную общей между отправителем и получателем сообщения, что означает создание нового канала связи.

Введем понятие системы процессов.

- *И-система процессов* — это совокупность процессов, которая:
  - завершается успешно, если все ее процессы завершаются успешно;
  - завершается неудачно, если хоть один из ее процессов завершается неудачно. При этом все другие свои процессы И-система насильственно завершает неудачно;
  - приостанавливается, если все ее процессы либо приостановились, либо уже завершились (успешно).
- *ИЛИ-система процессов* — это совокупность процессов, которая:
  - завершается неудачно, если все ее процессы завершаются неудачно;
  - завершается успешно, если хоть один из ее процессов завершается успешно. При этом все другие свои процессы ИЛИ-система насильственно завершает неудачно;
  - приостанавливается, если все ее процессы либо приостановились, либо уже завершились (неудачно).

Переменные языка Concurrent Prolog обладают рядом свойств, отличающих их от обычных логических переменных.

- Унификация защищенной переменной с непеременным термом приостанавливается и, тем самым, приостанавливает некоторый процесс.
- Если переменную, которую ждет какой-либо процесс, установить равной непеременному терму, то это должно привести к возобновлению этого процесса.

- Семантика Concurrent Prolog такова, что одна и та же переменная может быть с точки зрения одного процесса защищенной (только для чтения), а с точки зрения другого — просто переменной (общей с первым процессом).

2.5. Заключение

2.5.1. Выводы и рекомендации

Иногда во время этой работы здравый смысл подсказывал мне, чтобы я перестал навязывать учащимся свои идеи, но, к сожалению, гордость и обыкновенное самолюбие взяли верх.

Эдвард Йордон

Императивность компьютера вынуждает развивать методологии, сохраняя их основу императивной. Поэтому многочисленные появляющиеся методологии часто представляют собой развитие и привнесение методов императивной или структурной методологий.

Наибольший интерес программиста вызывают методологии, учитывающие все стадии жизненного цикла программного обеспечения. Таких полных методологий две: структурная и объектно-ориентированная. Именно на эти две методологии мы будем обращать первоочередное внимание при дальнейшем рассмотрении.

В качестве основного фактора, влияющего на выбор методологии, назовем класс, к которому относится решаемая задача. Для очень многих задач вполне пригодна классическая структурная методология. Выполняя декомпозицию такой задачи, мы рассматриваем структуру системы в терминах иерархии функций и передачи информации. В случае выявления в задаче сложной структуры объектов и многочисленных связей между ними будет вполне естественно обратиться к объектно-ориентированной методологии. Задачи больших объемов также удобно решать, используя объектно-ориентированную методологию. Именно она достаточно хорошо поддерживает два общих метода борьбы со сложностью систем: обеспечение независимости компонент системы и использование в системах иерархических структур.

2.5.2. Проблемы и перспективы развития

Нет таких проблем, которые нельзя было бы создать!

Анатолий Рас

Начнем с того, что на данный момент наблюдается упадок и забвение практически всех методологий, кроме объектно-ориентированной. Обратимся к истории. Если в конце 1980-х годов объектно-ориентированная методология была

известна только энтузиастам, то в середине 1990-х многие программисты стали к ней присматриваться, а с конца 1990-х не знать ее и не быть ее приверженцем стало просто неприлично. К сожалению, это привело к незаслуженной потере внимания ко многим другим важным и полезным методологиям.

Существует большое количество менее исследованных или менее популярных методологий. Почти все они в достаточно большой степени близки к основным, уже рассмотренным. Например, следующие четыре методологии представляют собой различные модификации структурного программирования.

- Методология *автоматного программирования* — это подход, предполагающий использование аппарата конечных автоматов [Шалыто 1998]. В основу проектирования программы закладывается алгоритм — конечный автомат в виде диаграммы состояний или таблицы последовательных переходов и выходов. Программисту удобно рассматривать автоматное программирование на основе конструкции `switch` (переключение), имеющейся в ряде языков программирования.
- Методология программирования, *управляемого потоками данных*, — это подход, заключающийся в том, что операции срабатывают не последовательно, а в зависимости от готовности данных [Чудаева 1992].
- Методология программирования, *управляемого событиями*, — подход, заключающийся в структуризации программного кода, основанного на идее наличия предопределенного множества поименованных событий. С каждым событием связана процедура (реакция на возникшее событие), автоматически вызываемая при возникновении события.
- Методология *доступ-ориентированного* программирования — подход, в котором функции с переменными связываются таким образом, что при доступе к переменной процедура будет вызываться автоматически [Bal, Grune 1994].

Методология *аспектно-ориентированного* программирования вводит понятие аспекта, которым является то свойство системы, которое не может быть явно реализовано в виде процедуры. Аспекты имеют тенденцию не быть элементами функциональной декомпозиции системы, но скорее быть свойствами, которые системно воздействуют на производительность или семантику компонентов. В этом аспекты противоположны компонентам, имеющим тенденцию быть единицами функциональной декомпозиции системы. Цель данной методологии — поддержать программиста в четком разделении компонентов и аспектов друг от друга, обеспечивая механизмы, которые сделают возможным абстрагировать их и объединять для получения системы в целом [Шукла, Селлз 2002].

Особняком идут некоторые перспективные нестандартные методологии. Например, методология *нейросетевого* программирования — подход, заключающийся в том, что на основе знаний, полученных от экспертов, создается программа на нейронном языке программирования, которая затем компилируется в эквивалентную нейронную сеть из аналоговых нейронов.

2.5.3. Рекомендации по литературе

Люди перестают мыслить, когда перестают читать.

Д. Дидро

Особенности императивного программирования изложены в огромном количестве книг. Наиболее систематично они приведены в работе "Универсальные языки программирования. Семантический подход" [Калинин, Мацкевич 1991].

Одна из лучших книг по объектно-ориентированной методологии — "Объектно-ориентированный анализ и проектирование с примерами приложений на C++" [Буч 1998]. Решения типичных задач объектно-ориентированного программирования удачно изложены в работе "Приемы объектно-ориентированного программирования. Паттерны проектирования" [Гамма, Хелм, Джонсон, Влиссидес 2001].

Методология функционального программирования подробно изложена в работах "Функциональное программирование" [Филд, Харрисон 1993] и "Функциональное программирование. Применение и реализация" [Хендерсон 1983]. В сети Интернет существует хорошее введение в функциональные языки Филиппа Уэдлера (Philip Wadler) (**http://cm.bell-labs.com/cm/cs/who/wadler/guide.html**).

Основные идеи методологии логического программирования изложены в сборнике статей "Логическое программирование" [Логическое 1988]. Элементы практического программирования на языке Prolog можно найти в книге [Бабаев, Герасимов, Косовский, Соловьев 1992].

Представление о ранних работах в области программирования в ограничениях можно получить из книги "Инструментальная система программирования ЕС ЭВМ (ПРИЗ)" [Кахро, Калья, Тыугу 1988]. Современное видение программирования в ограничениях изложено у Романа Бартака (Roman Bartak) (**http://kti.mff.cuni.cz/~bartak/constraints/**).

Одной из наиболее известных работ в области структурной методологии является статья "Заметки по структурному программированию" [Дейкстра 1975]. Методы структурного программирования подробно рассмотрены в книге "Теория и практика структурного программирования" [Лингер, Миллс, Уитт 1982]. Практику структурного программирования можно изучать по книге "Алгоритмы + структуры данных = программы" [Вирт 1985]. Философия визуального структурного программирования подробно изложена в работе [Паронджанов 1999].

Две книги являются хорошим введением в методологию параллельного программирования: "Программное обеспечение параллельных процессов" [Трахтенгерц 1987] и "Параллельные вычислительные системы" [Корнеев 1999].

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Языки

программирования" (Programming languages). Основные разделы этой темы:

- обзор языков программирования;
- объектно-ориентированное программирование;
- функциональное программирование.

❒ Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет следующие дисциплины специализации:

- параллельное программирование;
- функциональное программирование;
- рекурсивно-логическое программирование.

2.5.4. Вопросы и задания

1. Покажите, как в результате исторического развития появились современные методологии программирования.
2. Оцените преимущества и недостатки выбора каждой из методологий, принимая во внимание такие характеристики, как используемая память, время (как компьютерное, так и время программиста), безопасность и выразительная сила.
3. Опишите философию объектно-ориентированной методологии и понятий инкапсуляции, наследования и полиморфизма.
4. Спроектируйте, реализуйте, протестируйте и отладьте небольшие программы на объектно-ориентированном языке.
5. Объясните, как механизм классов поддерживает инкапсуляцию, наследование и полиморфизм.
6. Опишите сильные и слабые стороны функциональной методологии программирования.
7. Спроектируйте, реализуйте, протестируйте и отладьте несколько программ с использованием функциональной методологии.

ЛИТЕРАТУРА

Правду сказать, мы знаем жизнь только по литературе. Разумеется, за исключением тех, кто не знает литературы.

Станислав Ежи Лец

1. *Бабаев, Герасимов, Косовский, Соловьев 1992* — Бабаев И. О., Герасимов М. А., Косовский Н. К., Соловьев И. П. Интеллектуальное программирование. Турбо Пролог и Рефал-5 на персональных компьютерах. — СПб.: Изд-во С-Петербургского университета, 1992.

2. *Буч 1998* — Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. — М.: Бином, СПб.: Невский диалект, 1998.

3. *Вирт 1985* — Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985.

4. *Гамма, Хелм, Джонсон, Влиссидес 2001* — Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного программирования. Паттерны проектирования. — СПб.: Питер, 2001.

5. *Дейкстра 1975* — Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975, с. 7—97.

6. *Дейкстра 1978* — Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.

7. *Калинин, Мацкевич 1991* — Калинин А. Г., Мацкевич И. В. Универсальные языки программирования. Семантический подход. — М.: Радио и связь, 1991.

8. *Кахро, Калья, Тыугу 1988* — Кахро М. И., Калья А. П., Тыугу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). — М.: Финансы и статистика, 1988.

9. *Корнеев 1999* — Корнеев В. В. Параллельные вычислительные системы. — М.: Нолидж, 1999.

10. *Кун 1977* — Томас Кун. Структура научных революций. — М.: Прогресс, 1977.

11. *Лингер, Миллс, Уитт 1982* — Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. — М.: Мир, 1982.

12. *Логика 1990* — Логика и компьютер. Моделирование рассуждений и проверка правильности программ. — М.: Наука, 1990.

13. *Логическое 1988* — Логическое программирование. — М.: Мир, 1988.

14. *Паронджанов 1999* — Паронджанов В. Д. Как улучшить работу ума. — М.: Радио и связь, 1999.

15. *Тузов 1990* — Тузов В. А. Языки представления знаний. — Л.: Изд-во ЛГУ, 1990.

16. *Трахтенгерц 1987* — Трахтенгерц Э. А. Программное обеспечение параллельных процессов. — М.: Наука, 1987.

17. *Филд, Харрисон 1993* — Филд А., Харрисон П. Функциональное программирование. — М.: Мир, 1993.

18. *Хендерсон 1983* — Хендерсон П. Функциональное программирование. Применение и реализация. — М.: Мир, 1983.

19. *Хювенен, Сеппянен 1986* — Хювенен Э., Сеппянен Й. Мир Лиспа. В 2-х т. — М.: Мир, 1990.

20. *Чудаева 1992* — Чудаева О. В. Вычисления, управляемые потоками данных: модели, машины, языки (проблемы и перспективы развития). — Новосибирск: Изд-во ВЦ СО РАН, 1992.

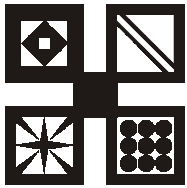
21. *Шалыто 1998* — Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. — СПб.: Наука, 1998.

22. *Шукла, Селлз 2002* — Шукла Д., Селлз С. Ф. К. АОП: Более эффективная инкапсуляция и повторное использование кода // MSDN Magazine/Русская редакция. 2002. Спецвыпуск №1.

23. *Язык 1988* — Язык Пролог в пятом поколении ЭВМ: Сб. статей 1983—1986 гг. / Сост. Н. И. Ильинский. — М.: Мир, 1988.

24. *Bal, Grune 1994* — Henri E. Bal, Dick Grune. Programming Language Essentials. Addison-Wesley Publishing Company, 1994.

25. *Hoare, Jifeng 1998* — C. A.R. Hoare, He Jifeng. Unifying Theories of Programming. Prentice Hall International, 1998.



Глава 3

Технологии программирования

Прочитав эту главу, вы получите ответы на следующие вопросы:

- Что такое жизненный цикл программ? Каковы основные технологические подходы к ведению цикла?
- Что такое классические и стандартные технологические процессы?
- Как формируются технологические стадии и в чем заключается порядок их прохождения?
- Что представляют собой технологии коллективной разработки программ?
- Как добиться создания качественного программного обеспечения?

3.1. Введение в технологии программирования

3.1.1. Основные понятия и определения

Если ты даешь собаке плохое имя, то лучше ее сразу пристрелить.
Английская пословица

Жизненный цикл программы — это весь период ее разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращением всех видов ее использования.

Технология программирования изучает технологические процессы и порядок их прохождения — стадии (с использованием знаний, методов и средств). Знания, методы и средства могут использоваться в разных процессах и, следовательно, технологиях.

Технологии удобно характеризовать в двух измерениях: вертикальном (представляющем процессы) и горизонтальном (представляющем стадии).

Процесс — совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Процессы состоят из набора действий, а каждое действие — из набора задач. Вертикальное измерение отражает

статические аспекты процессов и оперирует такими понятиями, как рабочие процессы, действия, задачи, результаты деятельности и исполнители.

Стадия — часть действий по созданию программного обеспечения, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными для данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяют в более крупные временные рамки, называемые фазами. Итак, горизонтальное измерение представляет собой время, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии, этапы, итерации и контрольные точки.

Технологический подход определяется спецификой комбинации стадий и процессов, ориентированной на разные классы программного обеспечения и на особенности коллектива разработчиков.

Простейшее представление жизненного цикла программы изображено на рис. 3.1.

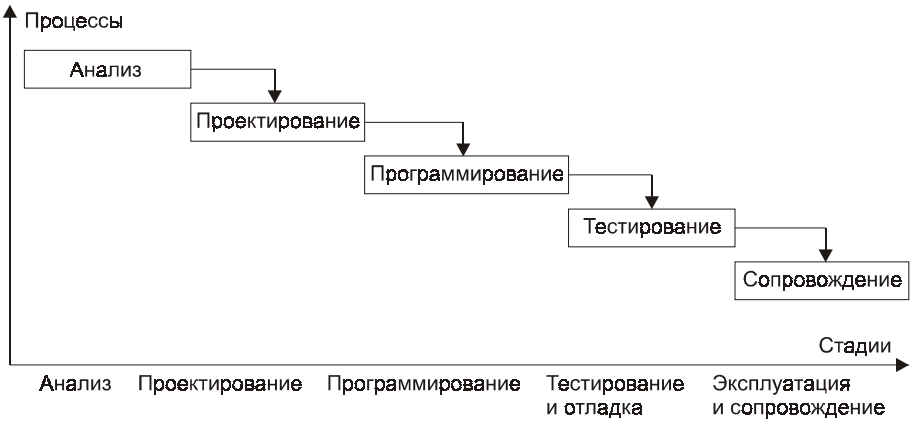


Рис. 3.1. Каскадный технологический подход к ведению жизненного цикла

Фактически, в данном случае на каждой стадии выполняется единственный процесс. Конечно, при разработке и создании больших программ такая схема недостаточно корректна (да и просто нереалистична). Однако мы можем взять ее за основу для многих других технологических подходов к ведению жизненного цикла.

3.1.2. История и эволюция

История учит тому, что из истории мы ничему не учимся.
Б. Шоу

В истории технологии программирования можно выделить три этапа.

- Осмысление опыта разработки больших систем, включающее понимание важности того, *как* ведется разработка программы, а не только *на*

каком языке программирования это делается. Этот этап также характеризуется проведением первых международных и национальных конференций (конец 60-х — 70-е годы XX века).

- 1968 г. — НАТО проводит первую конференцию по инженерии программирования (Software Engineering).
- 1975 г. — 1-я международная конференция IEEE.
- 1979 г. — 1-я Всесоюзная конференция по технологии программирования.

- Разработка новых технологических подходов (начало 70-х годов XX века — настоящее время).
  - 1973 г. — Дагласом Россом (Douglas Ross) разработана технология проектирования сложных систем SADT (Structured Analysis and Design Technique). Стандартизована под названием IDEF (Integrated DEFinition).
  - 1985 г. — Харланом Миллзом (Harlan Mills) сформулированы основные идеи технологии стерильного цеха.
  - 1995 г. — в октябре появилась первая пробная версия Унифицированного метода. Этот проект с 1996 года известен как UML (Unified Modeling Language). С точки зрения технологических подходов особый интерес представляет рациональный унифицированный процесс.
- Принятие стандартов на состав процессов жизненного цикла программного обеспечения и многочисленные попытки решить проблему качества программных продуктов (середина 80-х годов XX века — настоящее время).
  - 1985 г. — впервые утвержден стандарт жизненного цикла для проектирования программных систем (для систем военного назначения по заказам Министерства обороны США).
  - 1994 г. — в Великобритании создан международный консорциум, разрабатывающий на постоянной основе проекты стандартов и технологии быстрого создания приложений DSDM (Dynamic Systems Development Method).
  - 1995 г. — принят международный стандарт ISO 12207:1995 "Information Technology — Software Life Cycle Processes", регламентирующий состав процессов жизненного цикла программного обеспечения.

3.1.3. Классификации

3.1.3.1. Классификация технологических подходов

Выделим основные группы технологических подходов и укажем подходы для каждой из них.

Подходы со слабой формализацией

Эти подходы не используют явных технологий и их можно применять только для очень маленьких проектов, как правило, завершающихся созданием демонстрационного прототипа. К подходам со слабой формализацией относятся так называемые ранние технологические подходы, например подход "кодирование и исправление" *(см. разд. 3.5.1.1)*.

Строгие (классические, жесткие, предсказуемые) подходы

Данную группу подходов рекомендуется применять для средних, крупномасштабных и гигантских проектов с относительно ясными требованиями к системе и более-менее фиксированным объемом работ. Одно из основных требований к таким проектам — как можно большая предсказуемость. В эту группу входят следующие подходы.

- Каскадные технологические подходы.
  - Классический каскадный подход *(см. разд. 3.5.2.1)*.
  - Каскадно-возвратный подход *(см. разд. 3.5.2.2)*.
  - Каскадно-итерационный подход *(см. разд. 3.5.2.3)*.
  - Каскадный подход с перекрывающимися процессами *(см. разд. 3.5.2.4)*.
  - Каскадный подход с подпроцессами *(см. разд. 3.5.2.5)*.
  - Спиральная модель *(см. разд. 3.5.2.6)*.
- Каркасные подходы.
  - Рациональный унифицированный процесс *(см. разд. 3.5.3.1)*.
  - Модель процессов Microsoft Solution Framework *(см. разд. 3.5.3.2)*.

Формальные подходы

Формальные подходы предусматривают особые формальные требования к процессу создания программного обеспечения. Например, для генетических подходов требуются формальности, связанные с происхождением программы и дисциплиной ее создания. В данную группу входят следующие подходы.

- Генетические подходы.
  - Синтезирующее программирование *(см. разд. 3.5.4.1)*.
  - Сборочное (расширяемое) программирование *(см. разд. 3.5.4.2)*.
  - Конкретизирующее программирование *(см. разд. 3.5.4.3)*.
- Подходы на основе формальных преобразований.
  - Технология стерильного цеха *(см. разд. 3.5.5.1)*.
  - Формальные генетические подходы *(см. разд. 3.5.5.2)*.

Гибкие (адаптивные, легкие) подходы

Подходы этой группы рекомендуется применять для небольших или средних проектов в случае неясных или изменяющихся требований к системе. Команда разработчиков должна быть ответственной и квалифицированной, а заказчики должны быть согласны принимать участие в разработке. В данную группу входят следующие подходы.

- Ранние технологические подходы быстрой разработки.
  - Эволюционное прототипирование *(см. разд. 3.5.6.1)*.
  - Итеративная разработка *(см. разд. 3.5.6.2)*.
  - Постадийная разработка *(см. разд. 3.5.6.3)*.
- Адаптивные подходы.
  - Экстремальное программирование *(см. разд. 3.5.7.1)*.
  - Адаптивная разработка *(см. разд. 3.5.7.2)*.
  - Семейство технологических подходов Crystal *(см. разд. 3.5.7.3)*.
- Подходы исследовательского программирования.
  - Компьютерный дарвинизм *(см. разд. 3.5.8.1)*.
  - Фрагментарное программирование *(см. разд. 3.5.8.2)*.

3.1.3.2. Классификация технологических процессов

Мы будем рассматривать два набора (множества) технологических процессов. Первый набор — *классический*, включающий основные процессы, сложившиеся исторически в результате практического опыта разработки программного обеспечения. Второй набор — *стандартный*, т. е. основанный на стандарте ISO 12207:1995. Процессы классического набора фактически являются подмножеством стандартного, выступая там как процессы или действия процессов.

В классическом наборе выделим девять технологических процессов.

- Возникновение и исследование идеи *(см. разд. 3.2.1)*.
- Управление *(см. разд. 3.2.2)*.
- Анализ требований *(см. разд. 3.2.3)*.
- Проектирование *(см. разд. 3.2.3)*.
- Программирование *(см. разд. 3.2.4)*.
- Тестирование и отладка *(см. разд. 3.2.5)*.
- Ввод в действие *(см. разд. 3.2.6)*.
- Эксплуатация и сопровождение *(см. разд. 3.2.7)*.
- Завершение эксплуатации *(см. разд. 3.2.8)*.

Процессы жизненного цикла, определяемые международным стандартом ISO 12207 [ISO/IEC 12207:1995], делятся на три группы.

- Основные процессы.
  - Приобретение *(см. разд. 3.3.1.1)*.
  - Поставка *(см. разд. 3.3.1.2)*.
  - Разработка *(см. разд. 3.3.1.3)*.
  - Эксплуатация *(см. разд. 3.3.1.4)*.
  - Сопровождение *(см. разд. 3.3.1.5)*.
- Вспомогательные процессы.
  - Документирование *(см. разд. 3.3.2.1)*.
  - Управление конфигурацией *(см. разд. 3.3.2.2)*.
  - Обеспечение качества *(см. разд. 3.3.2.3)*.
  - Верификация *(см. разд. 3.3.2.4)*.
  - Аттестация *(см. разд. 3.3.2.5)*.
  - Совместная оценка *(см. разд. 3.3.2.6)*.
  - Аудит *(см. разд. 3.3.2.7)*.
  - Разрешение проблем *(см. разд. 3.3.2.8)*.
- Организационные процессы.
  - Управление *(см. разд. 3.3.3.1)*.
  - Создание инфраструктуры *(см. разд. 3.3.3.2)*.
  - Усовершенствование *(см. разд. 3.3.3.3)*.
  - Обучение *(см. разд. 3.3.3.4)*.

3.1.3.3. Классификация технологических стадий

Технологические стадии выделяются исходя из соображений разумного и рационального планирования и организации работ. Существует два основных варианта формирования промежутков времени, поддерживаемых технологическими подходами. В первом формируются *фазы*, отражающие крупные временные этапы. Например, начальная фаза, середина, кризис, окончание. Во втором варианте определяются стадии, отражающие названия классических процессов (или их подмножества, или надмножества), большая часть времени которых проходит в данной стадии. Пример взаимосвязи между стандартными процессами и стадиями показан на рис. 3.2.

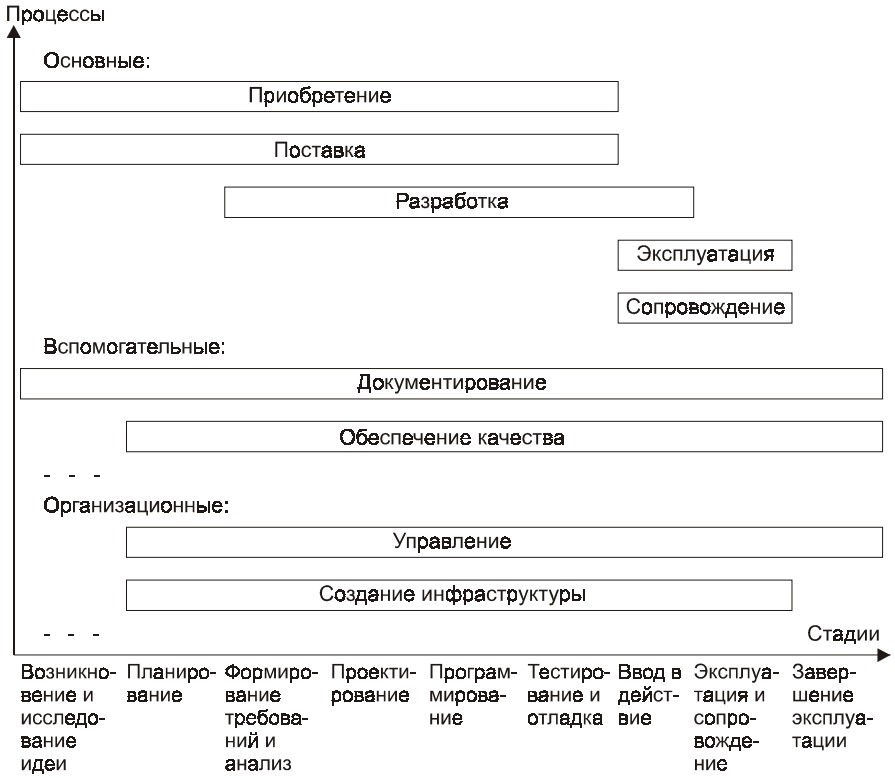


Рис. 3.2. Взаимосвязь между стандартными процессами и стадиями

3.2. Классические технологические процессы

Сначала неизбежно идут: мысль, фантазия, сказка. За ними следует научный расчет, и уже, в конце концов, исполнение венчает мысль.

К. Э. Циолковский

Начнем знакомство с "девятки" классических технологических процессов.

3.2.1. Возникновение и исследование идеи

Погоня за идеей — занятие столь же захватывающее, как и погоня за китом.

Генри Норрис Рассел

Этот классический процесс имеет следующие действия.

- Собственно возникновение и первичное исследование идеи, носящее максимально творческий и неформальный характер.
- Детальное исследование идеи, выработка концепции, постановка задачи, создание "одностраничного описания проекта" и разработка его

расширенной версии — технического задания. В случае заказного программного продукта техническое задание обычно готовится заказчиком работы.

- Экспертиза идеи специалистами, согласование технического задания с исполнителем и принятие решения о начале процесса планирования.

Заметим, что идея может привести либо к развитию уже существующего программного продукта, либо к созданию нового.

3.2.1.1. Возникновение идеи решения проблемы

Данный процесс обычно начинается с того, что у человека или небольшой группы людей возникает идея решения проблемы, которая:

- препятствует созданию или развитию программного продукта;
- приводит к ошибкам в программном продукте.

Наиболее интересными являются инновационные решения. *Инновация* — нововведение, изменяющее уже существующую результативную систему, как правило, с положительным эффектом или предлагающее новое решение некоторой проблемы в обозначенные время и месте [Косовский, Хитров 1997]. Инновация включает в себя и изобретения, и открытия, и просто новшества. Результативная инновация — это инновация, изменяющая уже существующее решение с улучшением некоторых его характеристик либо предлагающая к ранее не решенной проблеме решение, приносящее положительный эффект. Примеры результативных инноваций в программировании — создание глобальной сети Интернет и разработка интегрированной среды программирования.

В научном творческом поиске [Шевчук, Харахоркина, Журавлев, Армен 1998] отрицательный результат, как правило, также имеет существенное значение. С самого начала научного поиска следует понять, где проходит граница между знанием и незнанием. Научный поиск отличает определенное достоинство, заключающееся в верности критерию сдержанности и проверки идей, систематичности объяснений.

Поиск решения может включать эвристические правила. В качестве примеров укажем подход, предложенный Дьердем Пойа (Gyqrgy Ptlya) [Пойа 1957] для решения новых задач, и алгоритм решения изобретательских задач Генриха Сауловича Альтшуллера (**http://www.triz-ri.ru/triz/**). Роль интуиции в творческом поиске исследуется в книге [Грановская, Березная 1991].

Предложим несколько советов по организации поиска решения задачи [Косовский, Хитров 1997].

- Ждать, пока решение не придет в голову. Лишь при затянувшемся ожидании следует переходить к другим советам. Следует заметить, что особенно эффективен этот совет для ответа на следующие вопросы: "Что такое любовь?", "Что такое старость?".

- Следует понять — в чем смысл вопроса. Зачем вообще решать эту задачу? Здесь уместен принцип "выковыривания изюминки" — чтобы начать выковыривать изюминку, надо сначала найти булку с изюмом.
- Найти язык чертежей, формул, программ, на котором удается переформулировать задачу. Возможно, при переформулировке что-то станет яснее.
- Фиксировать внимание к произвольным мыслям и ощущениям.
- Выразить задачу на простом (детском), метафорическом языке. Поиск аналогий может привести к открытию новых фактов.

Эдвард де Боно (Edward de Bono) выделяет два типа мышления с соответствующими подходами решения задач (**http://www.edwarddebono.org**).

- *Вертикальное мышление*, которое начинает свой путь от выявления возможной причины той или иной проблемы или затруднения. Далее оно поступательно совершает один логический шаг за другим, пока не придет к некоторому решению.
- *Непоследовательное мышление*, которое может совершить скачок вперед, в сторону, в любое место вообще, чтобы неожиданно выработать решение. Примером непоследовательного мышления может служить развешивание зеркал на стенах рядом с дверями лифтов. Многочисленные жалобы жильцов на медленную скорость движения лифтов после этого прекратились. Зеркала дали возможность людям заняться собой, и им казалось, что они ждут лифт не так долго.

Заметим, что на этом этапе очень важно, чтобы бьющая через край энергия и заразительное возбуждение исследователей идеи не привели к небрежным результатам.

3.2.1.2. Постановка задачи

Теперь из идеи должно вырасти описание проекта. Все, что было предложено в первом действии, следует начинать формализовывать и документировать. Вклад в детальное исследование идеи могут делать любые заинтересованные люди.

В результате должна быть выработана концепция создаваемого или развиваемого программного продукта. Мы должны прояснить цель проекта, сформулировать требования к проекту и его результатам и выявить ограничения, которые существуют для достижения этих целей и выполнения этих требований.

Одностраничное описание проекта включает в себя несколько небольших разделов, кратко излагающих различные стороны предлагаемой идеи. Формально оно называется "одностраничным", подчеркивая краткость данной работы, но реально может занимать и две, максимум три страницы. Приведем пример описания проекта, связанного с разработкой документации для программного продукта.

Разработка руководства пользователя по OpenMP

Краткий обзор

Стандарт OpenMP представляет собой набор директив, добавление которых в код, написанный для последовательного исполнения, позволяет компилятору разделить программу на подзадачи для параллельного вычисления. Появление OpenMP стало первой стандартизацией параллельного исполнения программ.

Предлагается разработка документации "Руководство пользователя по OpenMP".

Введение

- Название проекта: Руководство пользователя по OpenMP.
- Дата подготовки документа: 19 августа 1998 версия 1.3.

Описание особенностей поставки

- Документ "Руководство пользователя по OpenMP" будет доступен как на сервере компании в сети Интернет, так и в бумажной копии.
- Поскольку в данной реализации проекта "Волхов" не будут включены все особенности спецификации OpenMP, необходимо четко указать, какие из них будут реализованы, а какие нет.
- Полная спецификация по OpenMP доступна в Интернете (**http://www.openmp.org/**).

Пользователи документа

Пользователь — опытный программист на языках C, C++, Pascal и FORTRAN, понимающий преимущества распараллеливания программ на многопроцессорных ЭВМ.

Сравнительный анализ

Многие разработчики компиляторов (например, компании SGI, Sun, IBM) уже включили поддержку OpenMP. Описание OpenMP они включают как отдельные главы в руководство пользователя по компиляторам с языков C, C++, Pascal и FORTRAN. С нашей точки зрения, лучшим решением будет иметь единый документ для всех компиляторов.

Описание технического процесса

1. Исследовать возможность импортирования спецификаций OpenMP непосредственно из документа.
2. Определить с помощью инженеров, реализующих поддержку OpenMP, какие особенности спецификации OpenMP войдут в состав текущей версии компилятора.
3. Разработать проект "руководства" и определить способ его распространения (совместно с группой маркетинга).

4. Проанализировать детали реализации, подготовить и включить в "руководство" соответствующие примеры программ.
5. Включить информацию о "руководстве пользователя" в документ "Новости компании Компилятор++".

Зависимости

- Выяснить возможность подключения инженеров-разработчиков для получения информации о деталях реализации и рецензирования документации.
- Выяснить право использовать фрагменты оригинальной спецификации OpenMP (это позволит сэкономить время, необходимое для разработки оригинальной части документации).

Список основных документов

Спецификация OpenMP.

Основные даты

- Завершение проектирования "руководства" — апрель 1998 г.
- Первый вариант полного "руководства" — август 1998 г.

Ресурсы

Ресурсы, необходимые для реализации данного проекта, представлены в табл. 3.1.

Таблица 3.1. *Ресурсы, необходимые для реализации проекта*

Функция	Ставка	Комментарии
Инженер-разработчик	0.1	Консультирующий инженер, реализующий OpenMP
Технический писатель	1.0	Подготовка документа
Руководитель проекта	0.3	Управление, разработка плана и содержания

3.2.1.3. Принятие решения о начале работы над проектом

Выясните сначала факты, а потом можете извращать их по своему усмотрению.

М. Твен

Практически всегда перед принятием решения проводится экспертиза идеи и проекта, который на ней основан. Специалисты должны в течение нескольких дней изучить и проанализировать идею (техническое задание). В их задачу входят три основных момента.

- *Подтвердить или поправить все предположения, на которых базируется проект.* На основании этих предположений будут делаться все дальнейшие построения.

- *Выявить и охарактеризовать все критические моменты в проекте.* Специалисты должны указать, что не предусмотрено и к каким последствиям это может привести.
- *Оценить время реализации проекта.* Для этого специалисты должны разобраться в задаче, определить, как можно выполнить декомпозицию будущего возможного продукта на компоненты, и оценить объемы этих компонент.

Итогом работы специалистов должна стать оценка осуществимости проекта.

И, наконец, завершиться данный процесс должен одной важнейшей формальностью — принятием решения о начале работы над проектом. Как минимум, это означает, что будет исполнен процесс планирования. Решение о начале работы может приниматься совместно заказчиком и исполнителем (в случае наличия конкретного заказчика) или управляющим комитетом компании (если продукт предназначен для широкого использования). Вопрос финансирования проекта (полного или частичного) может рассматриваться уже в рамках этого процесса, но чаще он принимается при завершении планирования.

3.2.2. Управление

Любую техническую проблему можно преодолеть,
имея достаточно времени и денег.
Закон Лермана

Вам никогда не будет хватать либо времени, либо денег.
Следствие из закона Лермана

Процесс управления длится почти весь жизненный цикл программного продукта, но поставили мы его в данном месте потому, что одним из важнейших действий управления является планирование, которое начинается вслед за принятием решения о том, что "проекту — жить".

3.2.2.1. Управление проектом

В условиях развитого рынка программного обеспечения и жесткой конкуренции важнейшее значение имеют: возможность получения максимальной прибыли, сроки разработки и тесное взаимодействие с другими коллективами разработчиков.

При профессиональной разработке программного продукта в крупных фирмах и компаниях предъявляются дополнительные требования к процессу разработки.

- Наличие формализованной модели для разработки программного продукта.
- Наилучшая расстановка приоритетов и ресурсов.

- Принятие ясных и документированных решений.
- Соблюдение многочисленных стандартов.
- Обеспечение эффективного взаимодействия между командами разработчиков, быстрое создание и вхождение в процесс разработки таких команд.
- Тесное взаимодействие с группами поддержки и продаж.

Каждая фирма строит свою деятельность, ориентируясь на основной критерий — прибыль. Получение прибыли связано с необходимостью производить для рынка товары (например, программный продукт) и услуги (например, консультации по программному продукту). Как правило, речь идет о производстве только той продукции, в которой нуждается потребитель.

Предпринимательская деятельность фирмы строится исходя из этой задачи, объединяющей пять направлений [Пашкус, Мисько 1991].

- Стратегию в области исследования и развития.
- Оперативную стратегию.
- Финансовую стратегию.
- Маркетинговую стратегию.
- Стратегию человеческих отношений.

Согласно этим направлениям управление можно поделить на три составные части: производственную, финансовую и маркетинговую.

Определим *управление (менеджмент)* как систему принятия решений в области управления фирмой и поделим всех служащих компании на четыре уровня.

- *Высший менеджмент* (президент фирмы, председатель совета директоров, исполнительный президент, вице-президенты). В его задачу входит определение генеральной линии компании. В области программного обеспечения особенно важно, чтобы высший менеджмент был "создателем видения", способным "гореть" идеями и вдохновить на их реализацию своих подчиненных. Он приносит в компанию осмысленность и вдохновение.
- *Управляющие среднего уровня* (менеджеры структурных подразделений). В их задачу входит координация и стратегическое планирование деятельности структурного подразделения.
- *Управляющие первичного звена* (менеджеры проектов, менеджеры-координаторы). Одна из важнейших задач этой категории управляющих — планирование на уровне программных проектов. Далее мы сконцентрируемся на изучении обязанностей только менеджеров проекта.

- *Инженеры* (внеменеджментовые служащие). Именно инженеры заняты разработкой и созданием программного продукта.

Лидерами команд разработчиков обычно являются два специалиста, тесно работающих вместе по всем направлениям разработки.

- *Управляющий (менеджер) проекта* — руководитель, обладающий организационными знаниями и опытом.
- *Технический руководитель (лидер) проекта* — инженер-технолог, обладающий техническими знаниями и опытом.

Заметим, что часто в небольших проектах и командах обе задачи исполняет один человек.

Обязанности управляющего

Выделим две группы обязанностей менеджера проекта. Первая группа выполняется совместно с лидером проекта, а вторая находится в исключительной ответственности менеджера.

Совместная деятельность менеджера и лидера проекта включает:

- планирование проекта;
- распределение работ;
- выбор наилучшей стратегии.

Исключительная ответственность менеджера заключается:

- в организации взаимосвязей внутри организации;
- во взаимодействии с вышестоящим начальством;
- в управлении сотрудниками. Менеджер ищет и привлекает лучших специалистов и экспертов для решения возникающих проблем;
- в руководстве проектом и контролем его выполнения. Менеджер отвечает за ежедневное руководство данным проектом. Хороший менеджер отводит различные проблемы от группы, он "гасит" нагоняи от заказчика или вышестоящего руководства;
- в том, чтобы проект отвечал требованиям заказчика. Руководитель должен держать заказчика в курсе всех событий проекта;
- в ответственности за поступление средств.

О том, что руководитель — это профессия

К сожалению, руководитель рассматривается многими как должность, а не как профессия. Система образования должна учитывать необходимость подготовки руководителей-профессионалов. Будущие руководители должны изучать психологию, социологию, управление персоналом, проводить деловые и ролевые игры, разбор ситуаций, моделирование профессиональной деятельности.

Еще раз обратимся к понятию "проект" и выделим четыре характеристики, делающих деятельность — проектом [Баранов 1998], а именно:

- *направленность на достижение конкретных целей.* Действительно проекты направлены на получение определенных результатов. Кстати, цели должны быть сформулированы так, чтобы всегда было ясно, что цель достигнута и проект можно заканчивать. Проект обычно предполагает наличие промежуточных целей, которые вместе образуют взаимосвязанный комплекс;
- *координированное выполнение взаимосвязанных действий.* Здесь следует еще раз вспомнить сложность разработки программного продукта. Взаимосвязи в проекте не всегда очевидны. Некоторые задания должны выполняться строго последовательно, а некоторые — строго параллельно;
- *ограниченная протяженность во времени* с определенным началом и концом. Проект имеет определенный срок. Иногда этот срок плавающий, например начинающийся от некоторой еще не определенной даты вступления договора в силу. Проект существует столько времени, сколько его требуется для получения конечного результата;
- *уникальность и важность.* Уникальность должна объяснять, почему надо создавать данный программный продукт, почему нельзя взять или купить что-то готовое. Элемент важности должен демонстрировать, почему разрабатываемый продукт нужен и важен заказчику, какие нужды и потребности заказчика он покрывает.

О том, что проектом не является

Простое несение обязанностей, деятельность без четких границ или бесцельное времяпровождение проектами не являются, поскольку не имеют определенных сроков и конкретных целей.

Таким образом, под *управлением проектом* будем понимать деятельность, направленную на реализацию проекта с максимально возможной эффективностью при заданных ограничениях по времени, денежным средствам и ресурсам, а также по качеству конечных результатов проекта.

Менеджер не должен позволять вмешиваться кому-либо другому в руководство проектом. На ежедневное руководство в среднем у него должно уходить 50% рабочего времени. Если менеджер тратит менее 50% времени на руководство, это означает, скорее всего, что он плохой руководитель и проект ждут серьезные проблемы.

О том, куда уходит время

Существует так называемый "феномен 5-го телефонного звонка". Он заключается в том, что нужный, но неизвестный пока человек находится в среднем после пятого телефонного звонка. Это объясняет, почему руководитель тратит так много времени на деятельность, связанную с осуществлением руководства.

Мотивация

А на фига?!
А. Вознесенский

Метод "Кнута и Пряника" — алгоритм, описанный в известной монографии Кнута [Кнут 2000] и позднее модифицированный русским программистом Пряником.
Программистский фольклор

Руководителям было издавна известно, что людей следует побуждать к действиям для достижения некоторого желательного результата. *Мотивация* — это процесс побуждения себя и других к деятельности для достижения некоторых целей [Овсянко 1999]. Исторически первый подход к мотивации имеет название *метод "кнута и пряника"*. Он заключался в побуждении либо под угрозой наказания, либо с использованием поощрения, либо комбинацией этих двух методов. Существует ряд психологических принципов, лежащих в основе теории мотивации.

- *Гедонизм* — заинтересованность людей делать то, что им приятно и уклоняться от того, что им неприятно. Менеджер может создать приятную обстановку на работе и таким образом мотивировать подчиненных.
- *Инстинкты* — автоматическая предрасположенность людей вести себя определенным образом. Некоторые люди находят удовольствие в том, чтобы ходить на работу, поскольку "рождены работать".
- *Концепция побуждений* — формирование привычек на основе действий, которые приводили к удовлетворяющему результату в прошлом.
- *Когнитивная теория* — рациональное поведение людей в соответствии с тем, что они думают о будущем. Менеджер может сформировать у работников впечатление о выгодности определенного поведения у подчиненных.

Потребность — осознанная недостаточность чего-либо. Именно потребности заставляют людей действовать определенным образом. Выделяют две основные группы потребностей.

- *Первичные*, являющиеся физиологическими по своей природе (например, потребность в пище).
- *Вторичные*, являющиеся психологическими (например, потребность в уважении).

Потребности можно удовлетворять с помощью вознаграждения. Менеджеры могут использовать внутренние вознаграждения (которые дает сама работа) и внешние (денежные выплаты и продвижение по службе).

Абрахам Маслоу (Abraham Maslow) в 40-х годах XX века разделил потребности на пять категорий.

- *Физиологические потребности*, являющиеся необходимыми для выживания (еда, жилище, секс).

- *Потребности в безопасности*, включающие потребности в защите от физических и психологических опасностей и уверенность в выполнении в будущем физиологических потребностей (пенсия).
- *Социальные потребности*, заключающиеся в привязанности, принадлежности к какой-либо общности, одобрении со стороны других, дружбе.
- *Потребности в уважении*, включающие внутренние (самоуважение, личные достижения) и внешние (статус, признание) факторы уважения.
- *Потребность в самовыражении*, заключающаяся в самореализации и росте как личности.

Перечисленные потребности образуют возрастающую иерархию. В каждый момент времени поведение человека определяется самой сильной из неудовлетворенных потребностей. Однако потребности высших уровней не мотивируют человека, пока не удовлетворены хотя бы частично потребности низших уровней. Менеджеры, работающие на международном уровне, должны иметь в виду, что относительная значимость различных потребностей людей может меняться в разных странах [Овсянко 1999].

3.2.2.2. Эволюция менеджмента

Наводить порядок надо тогда, когда еще нет смуты.
Лао Цзы

Основы менеджмента были заложены в начале XX века в европейской и американской науке. История менеджмента связана с Древним Египтом и философами античности. В настоящее время можно говорить об особенностях европейского, американского, японского и российского менеджмента.

Особенности европейского менеджмента

Европейский менеджмент имеет очень хорошую теоретическую базу, основанную на работах Макса Вебера (Германия), Анри Файоля (Франция), Кароля Адамецки (Польша) и многих других. Приведем несколько принципов управления Файоля [Психология 2000].

- *Дисциплина*, т. е. послушание и уважение к достигнутым соглашениям между фирмой и ее работниками. Дисциплина предполагает также справедливо применяемые санкции.
- *Вознаграждение* персонала, в том числе справедливая зарплата.
- *Справедливость* — сочетание доброты и правосудия.
- *Корпоративный дух*, т. е. гармония персонала, его сплочени.
- *Подчиненность личных интересов общим.* Интересы отдельного работника или группы не должны превалировать над интересами компании.

Особенности американского менеджмента

Менеджеры как социальная группа составляют в Америке около 15% работающего населения. Условием квалифицированного руководства считается соблюдение трех правил.

- *Анализируй* состояние инженеров, запросы рынка, процесс создания программного продукта.
- *Разделяй* обязанности среди подчиненных.
- *Контролируй* исполнение.

Американский менеджмент поощряет конкуренцию среди инженеров [Психология 2000]. Конкуренция представляется им необходимым условием социального и личностного развития. Это приводит к ряду негативных явлений, например сокрытию информации друг от друга и поиску более легкого, а не оптимального пути решения задачи. Наиболее слабым местом американского менеджмента является использование человеческого фактора в управлении.

Об особенностях американского менеджмента в области разработки программного обеспечения и о влиянии передовых информационных технологий на современный бизнес достаточно подробно рассказано в книге Билла Гейтса (Bill Gates) [Гейтс 2001].

О книге Билла Гейтса

Следует понимать, что данная книга предназначена для менеджеров, которые хотят познакомиться с передовыми методами ведения бизнеса. Программисты не найдут в ней практически ничего полезного для своей профессиональной работы. Интересную семантику заложили издатели в оформление обложки книги. Там крупными буквами написано Билл Гейтс и под именем нарисован значок `@`. Значок соответствует в английском языке предлогу "at" (являющемуся атрибутом адресов электронной почты), но традиционно произносится русскоязычными программистами как "собака". Это подчеркивает неоднозначное отношение программистов к Биллу Гейтсу.

Особенности японского менеджмента

На основе анализа деятельности японских компаний, Олстон сформулировал пять основных принципов японского менеджмента [Психология 2000].

- Рабочий *достаточно разумен*, он самостоятельно увеличивает производительность и качество своего труда. Три административных метода помогают претворить этот принцип в жизнь:
  - кружки качества;
  - практика стимулирования всех работников к совершенствованию профессиональных умений и навыков. Здесь очень интересно то, что японская традиция "минарай" — наблюдение за опытными

рабочими с целью освоения их навыков — очень близка русской традиции наставничества;

- практика перевода рабочих с одного места на другое. Работник повышает квалификацию, меняет специализацию и продвигается по служебной лестнице.

- Рабочий *стремится сделать свою работу лучше*. Методы реализации этого принципа:
  - пожизненный найм работников;
  - премирование рабочих в случае повышения прибыли фирмы. Заметим, что в неблагоприятной ситуации руководящий состав получает значительно более урезанную зарплату, чем рабочие.
- Рабочие *образуют "семью"*. Методы реализации принципа следующие:
  - работники фирмы оказывают новичку помощь, сочувствие и поддержку, ожидая от него в дальнейшем проявления такого же поведения по отношению к ним;
  - свободное время работники фирмы проводят вместе;
  - взаимные обязательства фирмы и рабочих. Если сотрудник фирмы вступает в брак или у него рождается ребенок, то он получает денежную прибавку, поскольку его расходы возросли.
- *Группа важнее отдельной личности*. Этот принцип основан на традиционной японской ценности — никто не должен быть эгоистичным и думать только о себе.
- Рабочие поддерживают *партнерство и сотрудничество* в производственных отношениях.

Особенности российского менеджмента

Понимание необходимости целенаправленной подготовки специалистов для управленческой деятельности на научной основе сложилось в нашей стране лишь в 60-е годы XX века. Российский менеджмент должен учитывать особенности национальной психологии, исторический опыт, традиции и ценности, которые веками складывались в России. Существует ряд приоритетных ценностей, на которые российский менеджмент должен опираться [Психология 2000].

- Социальные права личности.
- Честь и достоинство личности.
- Согласие и сотрудничество в обществе.
- Справедливость, основанная на понимании и принятии правовых норм.
- Профессионализм и мастерство, а также коллективизм, поддержка и взаимопомощь.
- Духовность и психическое здоровье в основе деловой и поведенческой практики.

3.2.2.3. Методы управления проектами

В основе методов управления проектами лежат методики сетевого планирования, разработанные в конце 50-х годов XX века [Филлипс, Гарсиа-Диас 1984]. С помощью этих методик руководитель проекта может:

- заблаговременно планировать работу над проектом и предвидеть возможные источники затруднений и задержки выполнения его в срок;
- планировать завершение работ в нужные сроки в соответствии с требуемой последовательностью выполнения заданий;
- координировать и контролировать выполнение работ для соблюдения календарного графика и завершения проекта в срок.

Следующие два метода были разработаны параллельно и независимо.

- *Метод Критического Пути* (МКП). Метод разработан фирмой "Дюпон де Немур" (Dupont de Nemours). Первоначально назывался методом Уолкера-Келли, по именам создателей. Позже получил название МКП (Critical Path Method — CPM).
- *Метод анализа и оценки программ ПЕРТ* (Program Evaluation and Review Technique — PERT). Метод был разработан корпорацией "Локхид" (Lockheed) и консалтинговой фирмой "Буз, Аллен энд Гамильтон" (Booz-Allen & Hamilton) и применен в военно-морских силах США для ускорения разработки баллистической ракеты "Поларис" для подводных лодок.

Для описания современных проектов, в которых связи между работами значительно сложнее, данные средства оказались недостаточными. Метод *сетевого планирования* описывает зависимости работ средствами теории графов [Романовский 1999].

Планирование проекта

Подчеркнем в виде неформальных требований необходимость наличия плана [Баранов 1998].

- План помогает создать ясное и четкое понимание, как будущие работы будут выполняться.
- План определяет роль каждого человека в исполнении проекта.
- План увязывает части работ вместе и позволяет видеть все взаимозависимости между разными частями работы.
- План — это точка отсчета для любых последующих изменений.
- С помощью плана мы видим, когда цель, поставленная перед проектом, достигнута и проект заканчивается.

Заметим, что первое действие процесса планирования — это принятие решения о начале планирования.

Планирование проекта включает определение ресурсов — человеческих, вычислительных и организационных — и составление "карты" задач и времени их выполнения. В стандартном наборе процессов планирование является одним из действий процесса управления. Общий подход к планированию выглядит так:

- построение списка задач;
- составление графиков выполнения работ;
- оценка затрат;
- выделение требуемых ресурсов;
- распределение ответственности;
- определение зависимостей между задачами;
- проведение персональных назначений на задачи;
- определение времени выполнения задачи;
- оценка рисков, связанных с конкретными задачами;
- выявление критических путей;
- создание инфраструктуры управления.

В рамках планирования управляющий решает также организационные задачи:

- анализ документации на полноту, содержание, аккуратность;
- назначение начальной и конечной дат работ;
- определение интерфейсов приложения и планирование работ по детальному проектированию интерфейсов.

Работа не может начаться раньше, чем будут выполнены все предшествующие ей работы. Продолжительность выполнения всего проекта не может быть меньше, чем сумма продолжительностей работ, входящих в любой путь от начального до заключительного события. Длиной пути назовем сумму продолжительностей работ, входящих в путь. *Критическим путем* будем называть путь наибольшей длины [Романовский 1999]. Наибольший риск в выполнении проекта будет иметь задержка в работах на критическом пути. Обратим внимание на то, что первоначально выявленный критический путь может внезапно смениться совсем другим. Это может обесценить многие принятые решения по управлению проектом.

Методики оценок времени и затрат

Существует несколько методик оценок времени и затрат для составления планов и расчетов [Баранов 1998].

- Сверху вниз по крупным блокам аналогичного проекта. Поскольку аналогичный проект пройден и мы знаем реальные затраты для него, то для нового проекта берем соответствующие цифры с некоторыми поправками.

- Снизу вверх по составленному графику работ. Данные по трудозатратам и времени берем от исполнителей.
- Методика норм работы. Учет ведется по нормам, определяемым как среднее значение времени, уходящее на данную работу в данной компании. Например, время, уходящее на разработку одной строки машинного кода.
- Методика исторических соотношений. Стоимость одного проекта относится к стоимости другого пропорционально отношению их объемов в некоторой степени `n`. Степень `n` — число, которое надо знать или подобрать опытным путем.
- Методика приближенных вычислений. Существует несколько расчетных формул для времен выполнения задач проекта. Введем три оценочных времени: оптимистичное (О), реальное (Р) и пессимистичное (П). Тогда время выполнения задач проекта можно вычислить:
  - по формуле трапеций: `(О+2*Р+П)/4`;
  - по формуле Симпсона: `(О+4*Р+П)/6`.

Рекомендации для оптимистов

Оптимистам после ряда провала сроков можно предложить "осторожную" формулу: $О \times \pi \times e$, где π и e — известные математические константы. Альтернативная рекомендация — изменение временного интервала у итогового числа на более крупный, например "недели" на " месяц".

Распределение работ

Распределение работ включает определение уровня квалификации для исполнителей задач, составление списка потенциальных участников проекта и "отображение" исполнителей на задачи. Типичной проблемой может быть сопоставление требуемым рабочим местам тех потенциальных исполнителей, чей уровень квалификации не полностью соответствует требуемому. Существует несколько эвристических правил для персональных назначений [Conger 1994].

- Назначать опытных и квалифицированных людей на наиболее сложные задачи в критическом пути. Людей с меньшим количеством опыта — на менее сложные задачи и т. д.
- Определить последовательность работ для каждого исполнителя.
- Не перегружать исполнителей задачами, для исполнения которых потребуется больше времени, чем есть в реальности.

О важности выбора

Одного и того же можно достичь разными способами. Предложение одного-единственного пути может говорить о профессиональной непригодности руководителя в условиях свободного рынка [Баранов 1998]. Он должен изложить не менее трех способов решения задачи и продемонстрировать понимание всех

этих способов. Только тогда можно обсуждать достоинства и недостатки каждого из этих подходов. И только тогда можно выбрать тот, который в данной ситуации наиболее пригоден.

Наконец, напомним еще одно правило: 80% работы выполняют 20% ее участников.

3.2.2.4. Современные подходы к управлению проектом

Став бригадиром, я упростил этот процесс до мыслимого предела.
В. Ерофеев. "Москва — Петушки"

Подход сети менеджеров по разработке программного обеспечения (SPMN)

Продолжающиеся непрерывные неудачи в крупных программных проектах заставили министерство обороны США сформировать подразделение, которое получило название Сеть менеджеров по разработке программного обеспечения (Software Program Managers Network — SPMN) (**http://www.spmn.com**). Для этой организации были сформулированы четыре главных цели ее работы.

- Внедрить лучшие практические навыки создания программного обеспечения.
- Дать возможность руководителям проектов сфокусировать свои усилия на разработке качественного программного обеспечения, нежели на следовании должностным инструкциям и формальным методикам, которые только ухудшали состояние проекта.
- Дать возможность руководителям проектов применять лучшие мировые практические навыки с учетом локальной корпоративной культуры.
- Дать возможность быстро изучить и внедрить эти навыки в свою работу с помощью соответствующих методик обучения и программных систем.

Одним из основных результатов работы данной организации стал документ, рекомендующий 16 основных практик (навыков), включающих 9 основных и 7 поддерживающих.

Перечислим основные практики.

- *Формальное управление рисками.* Рекомендуется постоянно вести и анализировать списки основных важнейших рисков. У сотрудников не должно быть никаких иллюзий о допустимости рисков.
- *Согласованность интерфейсов.* Интерфейсы — необходимая часть архитектуры. Чем позже будут определены соглашения об интерфейсах, тем больше вероятность того, что систему придется проектировать заново.
- *Формальные проверки проекта.* Экспертизы, проверки, сквозной контроль — все те действия, которые позволяют устранить ошибки как можно раньше.

- *Планирование и управление на основе метрик.* В основу планов и оценок должны быть положены числовые оценки — метрики, накопленные в предыдущих проектах.
- *Контроль качества на детальном уровне.* Промежуточные итоги должны подводиться еженедельно или даже ежедневно.
- *Свободный доступ к информации о ходе проекта.* Проект будет испытывать большие трудности, если менеджер скрывает от остальной команды информацию о состоянии проекта.
- *Отслеживание причин возникновения ошибок для достижения высокого качества.* Ошибки должны быть формально отслеживаемы, каждый случай обнаружения и устранения ошибки должен быть обязательно документирован.
- *Конфигурационное управление.* Использование систем управления исходными кодами, позволяющих отслеживать изменения и дающих возможность вернуться к предыдущим версиям.
- *Ответственность и подотчетность руководства перед сотрудниками.* Такая мотивация сотрудников может значительно повысить производительность труда программистов.

Правила И. С. Ашманова

Известным специалистом в области разработки программного обеспечения Игорем Станиславовичем Ашмановым сформулирован ряд правил, предназначенных для руководителей, которым волей судеб пришлось заниматься новым для себя делом — управлять тем или иным "программистским" проектом (**http://www.ashmanov.com/**). Многим опытным программистам свод этих правил может показаться набором простых и давно известных истин, но начинающие менеджеры программных проектов зачастую не знают самых простейших вещей.

Уже первое правило Ашманова утверждает: "Не бывает технических проблем. Бывают только человеческие, то есть организационные". При запуске проекта менеджер вступает, прежде всего, в человеческие отношения с коллегами и подчиненными. Здесь важно понимание двух следующих утверждений.

- Управление программистами *не является какой-то особенной магией.* Для этого обязательно нужно вообще уметь управлять людьми и проектами. Как и в любой отрасли, менеджеру достаточно знать некоторые особенности технологического процесса и не поддаваться "мифам индустрии". Всё остальное зависит от обычного умения менеджера наладить работу.
- Управление программистами *имеет особенности, осложненные мифами и иллюзиями вокруг программирования.* Основной причиной мифов является противоречие между очевидной интеллектуальностью и сложностью

работы с одной стороны, и совершенно обычными свойствами персонала и проектов с другой стороны. Эти мифы охотно поддерживаются разработчиками, но с опытом и знанием приходит независимость от мифов.

Одним из основных выводов из свода правил является понимание практической невозможности перевоспитания разработчиков в деловом духе. Однако этого и не требуется. Вместо перевоспитания нужно понимание мотивов и привычек, а также работающие процедуры планирования, исполнения и контроля, сохраняющие свободу мышления и оставляющие простор для творчества. Схемы мотивации также должны учитывать авторские мотивы и врожденный оптимизм разработчика. Например, из этого общего принципа легко выводится недопустимость штрафов за срыв сроков.

Ашманов приводит небольшой словарик высказываний, которые менеджер не должен воспринимать буквально. Например, "работаем" — обычный ответ разработчика на вопросы менеджера. Ответ помогает "отбить" большую часть запросов о ходе проекта. Сам по себе этот ответ не является криминалом, поскольку на самом деле в разработке бывают периоды упорной работы "от забора до обеда", когда результатов не видно. Но частое повторение этой формулы подозрительно, поскольку она может служить и для сокрытия уже обнаружившихся проблем со сроками и трудоемкостью, которые разработчик надеется решить сам, не доводя до начальства.

И, наконец, приведем ряд формальных правил Ашманова о том, как правильно запустить проект и закончить его. Подробные комментарии к правилам можно найти в статье "Правила Ашманова. Управление проектами" (**http://www.ashmanov.com/pap/ashrul2/i.phtml**).

- Решение о запуске проекта должны принимать ответственные лица.
- Для начала проекта нужно исполнить процедуру запуска проектов.
- Без энтузиаста любой проект мертв.
- Проект нельзя обсуждать без обоснования.
- Не бывает проекта без руководителя.
- Руководитель должен быть один.
- Проект нельзя запускать без плана, написанного лично руководителем проекта.
- Проект нельзя запускать без ресурсов.
- В начале проекта всегда бывает естественное торможение. Чтобы его преодолеть, нужны терпение и настойчивость.
- Вполнакала проекты не делаются.
- Только коррекцией в контрольных точках можно удержать проект от срыва.
- По окончании проекта нужно правильно переключить ответственность.
- После завершения проекта нужно запустить следующий "бессрочный" проект — проект поддержки.

3.2.3. Анализ требований и проектирование

3.2.3.1. Введение в анализ требований и проектирование

Анализ и проектирование — два достаточно близких и тесно связанных процесса. Они выполняют общую задачу, результатом которой должно стать четкое *представление* о системе, на основе которого будет создан программный код. Поэтому мы будем их рассматривать совместно, проводя разделение по методологиям: анализ и проектирование на основе структурной методологии и на основе объектно-ориентированной методологии.

Анализ требований — процесс жизненного цикла программы, во время которого требования заказчика уточняются, формализуются и документируются. Основной вопрос, который решается в это время, — "что должна делать будущая система?".

Проектирование — процесс жизненного цикла программы, во время которого исследуется ее структура и взаимосвязи элементов. Проектирование, как правило, является итерационным процессом. Основной вопрос, который решается в это время, — "как система будет удовлетворять полученным требованиям?".

> ***О представлении о системе***
>
> Обратим внимание на то, что свойства программы в общем случае окончательно выясняются лишь после завершения ее исполнения (к этому ведет принцип построения преобразователей предикатов). Это верно и по отношению к проектам. Надежное представление о системе складывается лишь в результате завершения ее разработки. Отсюда следует, что анализ требований и проектирование создают лишь некоторую иллюзию реальности (в части достаточности информации).

Проектирование должно проводиться на двух уровнях.

- Проектирование *архитектуры* (проектирование системы, проектирование "в большом").
- *Детальное* проектирование (проектирование модулей, проектирование "в малом").

Результатом анализа и проектирования должен стать проект, содержащий достаточное количество информации для реализации системы на его основе.

На сегодняшний день в профессиональном программировании применяются две основные методологии: структурная и объектно-ориентированная. На этапе анализа и проектирования нам важно принципиальное различие между этими методологиями, заключающееся в разных способах декомпозиции систем [Вендров 2000].

- *Структурная (функциональная)* декомпозиция рассматривает структуру системы в терминах иерархии функций и передачи информации.

- *Объектная* декомпозиция рассматривает структуру объектов и связей между ними, а также поведение системы в терминах обмена сообщений между объектами.

Далее мы перечислим основные методы структурного и объектно-ориентированного анализа и проектирования. Обратим внимание на то, что в основе многих объектно-ориентированных методов лежит структурный подход, которому придана объектно-ориентированная окраска.

В основе ведения процессов анализа и проектирования лежат методы, поддерживаемые языками и системами. Методы, объединенные в некоторую совокупность (комбинацию), образуют *подходы к анализу и проектированию*. Подходы имеют названия, среди которых много именных. Например, подход Йордона и ДеМарко (структурная методология) и подход Шлеер и Мелора (объектно-ориентированная методология).

3.2.3.2. Отступление "о спецификациях"

– Вовочка, ты уже покрасил окна?
– Да, осталось покрасить только рамы!

Анекдот о необходимости точных спецификаций

Спецификация — достаточно точное и достаточно полное описание задачи, которое человеку, участвующему в решении, написать, понять и прочесть легче, чем программу решения этой задачи на доступном ему языке программирования. Более коротко спецификацию можно определить как подробное описание некоторой работы, подлежащей выполнению.

Средства спецификации — любые средства получения или построения таких описаний.

Язык спецификации — рационально оформленный и синтаксически организованный набор таких средств.

В спецификациях программ имеет смысл выделить две существенно отличающиеся части.

- *Функциональные* спецификации, описывающие функцию программы.
- *Эксплуатационные* спецификации, описывающие скорость работы программы, используемые ресурсы, характеристики аппаратуры, специальные требования к надежности и безопасности.

Существует точка зрения об относительности понятия спецификации [Агафонов 1984]. Имея дело с одной и той же задачей, данные люди в данном окружении (включающем их знания, языки и т. п.) будут склонны считать спецификацией одно, а другие люди в другой обстановке — совсем другое. Например, "`x := x + 1`" будет лучшей спецификацией задачи "увеличить `x` на единицу" для человека, знакомого с оператором присваивания и его обозначением. Есть задачи (например, вычисление факториала), где никакая спецификация не отличается от программы.

Средства спецификации, которые допускают машинную обработку, мы будем классифицировать по уровню формализации и по способу представления [Деметрович, Кнут, Радо 1989].

По уровню формализации выделим три класса.

- *Словесные* спецификации, обработка которых может осуществляться обычным текстовым редактором.
- *Модельные (структурированные)* спецификации, которые предполагают построение схем, диаграмм и других информационных структур.
- *Формальные* спецификации, получаемые строгим формальным способом с использованием математических формализмов, которые обеспечивают полное определение семантики.

Существует два основных способа представления спецификаций.

- *Текстовое* представление, которое подходит для словесных и формальных спецификаций.
- Представление с помощью *информационных объектов*. Как правило, таким способом представляются модельные спецификации.

В завершении разговора о спецификациях отметим, что, пожалуй, нельзя привести примеры точных спецификаций, прочесть и понять которые, было бы проще, чем программу.

3.2.3.3. Отступление "об архитектуре"

Архитектура программного продукта — это его строение как оно видно (или должно быть видно) извне его, т. е. представление программного продукта как системы, состоящей из некоторой совокупности взаимодействующих подсистем [Жоголев 1996]. В качестве таких подсистем обычно выступают отдельные программы.

Следующее определение принадлежит Роберту Криспену (Robert Crispen) (**http://www.sei.cmu.edu/architecture/definitions.html**): *Архитектура* состоит из (а) стратегии разделения и (б) стратегии объединения. Стратегия разделения заключается в делении монолитной системы на дискретные, полностью покрывающие ее части (или компоненты). Стратегия объединения заключается в явном определении интерфейсов между этими компонентами.

И, наконец, еще одно определение содержится в работе [Буч, Рамбо, Джекобсон 2000]: *Архитектура* — это совокупность существенных решений, касающихся:

- организации программной системы;
- выбора структурных элементов, составляющих систему, и их интерфейсов;
- поведения этих элементов, определенного в процессе взаимодействия с другими элементами;

- составления из этих структурных и поведенческих элементов все более и более крупных подсистем;
- архитектурного стиля, направляющего и определяющего всю организацию системы: статические и динамические элементы, их интерфейсы, кооперации и способ их объединения.

Выделяют следующие классы архитектур программных продуктов [Майерс 1980], такие как:

- цельная (монолитная) программа;
- комплекс автономно выполняемых программ;
- слоистая программная система;
- коллектив параллельно выполняемых программ.

Можно определить четыре структуры, которые в совокупности описывают программную архитектуру [Ахтырченко, Леонтьев 2001].

- *Логическая (концептуальная) структура*, включающая множество абстракций, необходимых для описания функциональных требований к системе на абстрактном уровне (уровне, не затрагивающем вопросы реализации). Данная структура строится на основе анализа проблемной области.
- *Модульная структура*, определяющая организацию программного средства как совокупности модулей — программных единиц, с которыми могут быть соотнесены рабочие задания, выполняемые членами проектной команды.
- *Процессная структура*, описывающая поведение системы во время ее исполнения.
- *Физическая структура*, определяющая отображение элементов программного средства на аппаратное обеспечение.

Кроме этих четырех часто используются и другие структуры.

- Структура вызовов.
- Структура потоков данных.
- Структура потоков управления.
- Структура использования.
- Структура классов.

Практически все технологические подходы содержат рекомендации по моделированию большинства перечисленных структур.

Мэри Шоу (Mary Shaw) и Дэвид Гарлан (David Garlan) предложили использовать каталог архитектурных стилей [Shaw, Garlan 1996].

- Независимые компоненты:
  - взаимодействующие процессы;
  - системы с событиями: неявный вызов и явный вызов.

- Поток данных:
  - пакетно-последовательный;
  - каналы и фильтры.
- Централизованные данные:
  - хранилище;
  - классная доска.
- Виртуальные машины:
  - интерпретатор;
  - системы с правилами.
- Вызов с возвратом:
  - главная программа и подпрограммы;
  - объектно-ориентированный;
  - слойный.

Идентификация и документирование таких архитектурных стилей позволяет программистам использовать их в качестве стартовой точки. Хорошие примеры архитектурных паттернов (шаблонов) и каркасов проектирования приведены в книге [Гамма, Хелм, Джонсон, Влиссидес 2001].

> ***Об использовании паттернов***
>
> Впрочем, паттерны проектирования нередко используются там, где без них можно было бы обойтись. Например, в ряде случаев вместо паттерна "стратегия" можно просто использовать оператор `if`. Заодно заметим, что многие паттерны (как и архитектура вообще) платформенно-зависимы, и это нужно четко учитывать при разработке программ.

3.2.3.4. Отступление "о классификации всего сущего"

Женщина.
Исихара Икидаэ (XI век). Из цикла "Стихотворения из одного слова"

Во многих ситуациях (например, для разработки диаграммы "сущность-связь") нам потребуется выделить совокупность сущностей из окружающего нас мира. Следует разобраться, как удобнее классифицировать окружающий нас мир.

Большинство окружающих нас объектов относится к категориям, рассмотренным в книге [Шлеер, Меллор 1993].

- *Реальные объекты* — абстракции фактически существующих предметов в физическом мире.
- *Роли* — абстракции цели или назначения человека, части оборудования или организации.
- *Инциденты* — абстракции чего-то произошедшего или случившегося.
- *Взаимодействия* — объекты, получающиеся из отношения между другими объектами.

Образование иерархий широко используется человеком как стратегия организации информации. Иерархия предпочтений, выявленная Килем в ходе психологических экспериментов с детьми, представлена на рис. 3.3.

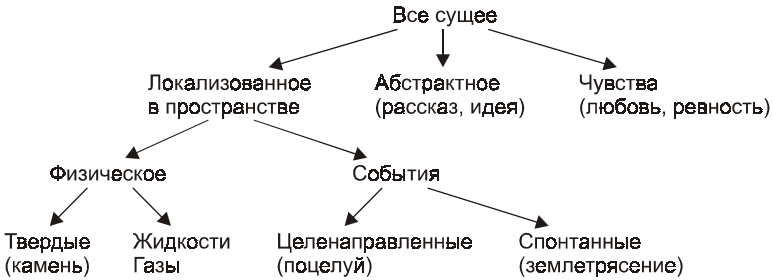


Рис. 3.3. Одна из классификаций всего сущего

Классифицировать можно даже ненаблюдаемые объекты. Их можно разбить на четыре категории, расположенные в порядке убывания научного интереса, который они представляют [Физики 1968].

- Явления, ненаблюдаемые по определению (например, невидимый свет).
- Явления, ненаблюдаемые в принципе (например, абсолютная скорость).
- Явления, ненаблюдаемые в природе (например, потомство от стерильных кроликов).
- Явления, ненаблюдаемые в обществе воспитанных людей (например, ковыряние в носу).

3.2.3.5. Проектирование архитектуры (проектирование "в большом")

> Верите ли, это бредовое сооружение Ирода, — прокуратор махнул рукою вдоль колоннады, так что стало ясно, что он говорит о дворце, — положительно сводит меня с ума. Я не могу ночевать в нем. Мир не знал более странной архитектуры.
>
> *Михаил Булгаков. "Мастер и Маргарита"*

Структурная методология

Проектирование архитектуры (проектирование "в большом") для структурной методологии включает следующие основные методы:

- метод нисходящего проектирования;
- метод восходящего проектирования;
- метод расширения ядра.

В случае структурной методологии в качестве модульной структуры программы принято использовать древовидную структуру. В узлах такого дерева размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей.

Методы нисходящего и восходящего проектирования имеют достаточно большое количество разновидностей и модификаций (например, зависящих от того, начинается кодирование до окончательного завершения проектирования или нет).

Метод *нисходящего проектирования* представляет собой подход функциональной декомпозиции на основе двух стратегий.

- *Пошагового уточнения*, при котором на каждом следующем этапе декомпозиции определяются модули очередного, более низкого уровня.
- *Анализа сообщений*, при котором анализируются потоки данных, обрабатываемые модулями.

Метод *восходящего проектирования* — подход, при котором в первую очередь определяются вспомогательные модули, которые потребуются для проектируемой программы.

Метод *расширения ядра* — подход, при котором основное внимание уделяется выявлению множества вспомогательных модулей, а не определению функции всей программы в целом.

Объектно-ориентированная методология

Проектирование архитектуры для объектно-ориентированной методологии включает следующие основные методы [Шлеер, Меллор 1993]:

- метод проектирования предметных областей;
- метод наведения мостов.

Метод проектирования предметных областей заключается в выделении предметной области системы с точки зрения пользователя. *Предметная область* — это отдельный реальный, гипотетический или абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соответствии с характерными для домена правилами и линиями поведения.

Метод наведения мостов заключается в том, что одна предметная область использует механизмы и возможности, обеспечиваемые другой предметной областью. Мост между двумя предметными областями представляет собой набор предложений (с точки зрения пользователя) и набор требований (с точки зрения исполнителя).

3.2.3.6. Проектирование модулей (проектирование "в малом")

Структурная методология

Перечислим основные методы проектирования модулей для структурной методологии и рассмотрим подробно некоторые из них.

- Диаграммы "сущность-связь".
- Структурные карты.
- Диаграммы деятельности.

- Диаграммы Варнье-Орра.
- Диаграммы переходов состояний.
- Блок-схемы.
- Схемы экранов.
- Псевдокод.

Диаграмма "сущность-связь" (Entity-Relation Diagram — ERD) предназначена для разработки моделей данных и обеспечивает стандартный способ определения данных и отношений между ними.

При создании таких моделей рассматривают несколько понятий.

- *Сущность* — это такая абстракция множества объектов реального мира, в которой все предметы этого множества имеют одинаковые характеристики и согласованы с одним и тем же набором правил и линий поведения. Выделяют следующие типы сущностей:
  - *независимая сущность* представляет независимые данные, которые всегда присутствуют в системе. Отношения с другими сущностями у нее могут отсутствовать;
  - *зависимая сущность* представляет собой данные, зависящие от других сущностей в системе. Она всегда имеет отношения с другими сущностями;
  - *ассоциированная сущность* представляет собой данные, которые ассоциируются с отношениями между двумя и более сущностями.
- *Отношение* связывает две или большее количество сущностей. Отношение всегда выражается действием, и его именуют с помощью глагола. Выделяют следующие типы отношений:
  - *неограниченное отношение* представляет собой безусловное отношение, которое существует до тех пор, пока существуют относящиеся к делу сущности;
  - *ограниченное отношение* представляет собой условное отношение между сущностями;
  - *существенно-ограниченное отношение* используется, когда соответствующие сущности взаимозависимы в системе.
- *Связи* используются для идентификации требований, в соответствии с которыми сущности вовлекаются в отношения. Каждая связь соединяет сущность и отношение и может быть направлена только от отношения к сущности. Пара значений связей, принадлежащих одному и тому же отношению, определяет тип этого отношения. Для большинства приложений достаточно использовать следующие типы отношений:
  - 1:1 (один-к-одному);
  - 1:M (один-ко-многим);
  - M:N (многие-ко-многим).

- *Атрибут* — абстракция одной характеристики, которой обладают все абстрагируемые как объект сущности. Атрибуты бывают:
  - *описательные* — представляющие факты, внутренне присущие каждому экземпляру объекта;
  - *указательные* — для дачи имени или обозначения экземпляру;
  - *вспомогательные* — для связи экземпляра одного объекта с экземпляром другого.

Разработанные диаграммы "сущность-связь" далее практически всегда следует преобразовывать в реляционную модель, которая допускает только унарные и бинарные связи. Подробно о реляционной модели рассказано в *разд. 4.4.3.2.*

Псевдокод (pseudocode) — метод применения стилизованного естественного языка для описания структуры управления программы. Конструкции такого языка обычно близки к конструкциям блочных структурных языков программирования. Псевдокод состоит из таких предложений, которые могла бы выполнить ЭВМ, но которые содержат вместо ключевых слов, зависящих от используемого языка программирования, фразы в свободной форме.

Объектно-ориентированная методология

Перечислим основные методы проектирования модулей для объектно-ориентированной методологии и рассмотрим подробно некоторые из них.

- Диаграммы кооперации.
- Диаграммы компонентов.
- Диаграммы развертывания.

Диаграмма кооперации (collaboration diagrams) предназначена для графического представления всех структурных отношений между объектами, участвующими во взаимодействии. Кооперация служит для обозначения множества взаимодействующих с определенной целью объектов. Кооперация может быть представлена на двух уровнях.

- *На уровне спецификации*, показывая роли классификаторов и ассоциаций в рассматриваемом взаимодействии.
- *На уровне примеров*, указывая экземпляры и связи, образующие отдельные роли в кооперации.

Объекты на диаграмме располагаются в виде вершин графа. Связи соединяют вершины дугами. Связи могут быть аннотированы сообщениями, которые объекты принимают и посылают.

Диаграмма компонентов (component diagram) описывает особенности физического представления системы. Она позволяет определить архитектуру разрабатываемой системы, устанавливая зависимости между компонентами.

Диаграмма развертывания (применения, размещения) (deployment diagram) отображает физические взаимосвязи между программными и аппаратными компонентами системы.

3.2.3.7. Методы анализа и построения спецификаций

Лифчики женские.

Спецификация на ценнике в магазине

Структурная методология

Перечислим основные методы ведения структурного анализа и рассмотрим подробно некоторые из них.

- Диаграммы потоков данных.
- Диаграммы потоков управления.
- Таблицы решений.
- Сети Петри.
- Диаграммы зависимости.
- Диаграммы декомпозиции.
- Диаграммы функционального моделирования.

Диаграмма потоков данных (Data Flow Diagram — DFD) — информационная модель, основные компоненты которой — это различные потоки данных, переносящие информацию от одного модуля к другому.

Компонентами диаграммы потоков данных являются:

- *внешние сущности* — материальный предмет или физическое лицо, представляющее собой источник или приемник информации;
- *процессы* — совокупность операций по преобразованию входных данных в выходные в соответствии с определенным алгоритмом;
- *накопители данных* — абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь;
- *поток данных* — механизм, определяющий информацию, передаваемую через некоторое соединение от источника к приемнику.

Модель системы характеризуется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи пользователю.

Диаграмма функционального моделирования (Structured Analysis and Design Technique — SADT) — модель, состоящая из диаграмм, фрагментов текста и глоссария, имеющих ссылки друг на друга.

В состав диаграммы входят:

- *блоки*, изображающие активность моделируемой системы;
- *дуги*, связывающие блоки вместе и изображающие взаимодействия и взаимосвязи между ними. Место соединения дуги с блоком определяет тип интерфейса:
  - *управляющая информация* входит в блок сверху;
  - *входная информация*, подвергающаяся обработке, показана с левой стороны блока;
  - *результаты* (выходная информация) показаны с правой стороны;
  - *механизм* (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу.

Объектно-ориентированная методология

Перечислим основные методы ведения объектно-ориентированного анализа и рассмотрим подробно некоторые из них.

- КОК-карты (класс-ответственность-кооперация).
- Диаграммы вариантов использования.
- Диаграммы классов.
- Диаграммы состояний.
- Диаграммы деятельности.
- Диаграммы последовательности.

Карты класс-ответственность-кооперация (class-responsibility-collaboration) предназначены для описания классов. Первоначально они были созданы для обучения объектному языку (**http://c2.com/doc/oopsla89/paper.html**). Поясним смысл двух терминов, входящих в название метода.

- *Ответственность* — это высокоуровневое описание функций, которые выполняет класс. Здесь применяется идея отказаться от описания конкретных элементов данных и процессов и вместо этого несколькими предложениями описать ответственность класса.
- *Кооперация* — ссылка на другие классы, с которыми необходимо кооперироваться для реализации функций. С помощью кооперации можно получить представление о связях между классами.

Этот метод способствует более оживленному обсуждению модели и концентрирует внимание на ответственностях, позволяя понять особенности поведения каждого класса.

Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов не содержит информации о временных аспектах функционирования системы.

Класс служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношением с объектами из других классов. Между классами существуют четыре типа отношений.

- *Ассоциации*, представляющие некоторое отношение или связь.
- *Обобщения*, представляющие иерархическое строение и наследование классов.
- *Зависимости*, когда изменение одного элемента может потребовать изменения другого.
- *Агрегации* (часть-целое), когда один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Объект — это отдельный экземпляр класса, который создается на этапе выполнения программы. Объект имеет имя и конкретные значения атрибутов.

Диаграмма объектов (object diagram) может понадобиться, если требуется рассмотреть взаимосвязи между отдельными объектами.

Обратим внимание на то, что практически во всех объектно-ориентированных подходах есть возможность группировать классы в компоненты более высокого уровня (например, предметные области или пакеты).

Диаграмма вариантов использования (use case diagram) — сценарий, позволяющий лучше понять требования к системе. В состав диаграммы входят:

- *вариант использования* (use case) — типичное взаимодействие пользователя и системы. Оно охватывает некоторую очевидную для пользователя функцию, решая некоторую дискретную задачу;
- *действующее лицо* (*актер* — actor) — любая внешняя по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функции и возможности;
- *интерфейс* — спецификация параметров системы, которые видимы извне без указания их внутренней структуры.

Диаграмма состояний (statechart diagram) отражает модель поведения объектов в реальном или абстрактном мире. Все объекты имеют некоторый срок жизни:

- сначала они появляются или создаются;
- затем эволюционируют, проходя через определенные стадии существования;
- и, наконец, исчезают или умирают.

Для представления этого жизненного цикла предлагается модель состояний, состоящая из:

- множества *состояний*. Состояние — положение объекта, в котором применяется определенный набор правил, линий поведения, предписаний и физических законов;

- множества *событий*. Событие — абстракция инцидента или сигнала в реальном мире, который сообщает о перемещении чего-либо в новое состояние;
- множества *правил переходов*. Правила определяют, какое новое состояние достигается, когда с объектом в данном состоянии происходит некоторое событие;
- множества *действий*. Действие — это деятельность или операция, которая должна быть выполнена экземпляром, когда он переходит в некоторое состояние.

Диаграмма деятельности (activity diagram) предназначена для детализации особенностей алгоритмической и логической организации, выполняемой системой операций. Каждое действие расчленяется на фундаментальные процессы. Управление на диаграмме осуществляется либо явно — через потоки управления, либо неявно — через определяемые потоки данных. Процессы могут быть объединены в четыре основные группы.

- Процессы *доступа*, единственная цель которых заключается в получении доступа к элементам одного архива данных объекта.
- *Генераторы событий*, создающие событие как результат работы.
- Процессы *преобразования*, основной целью которых является вычисление или преобразование данных.
- Процессы *проверки*, которые проверяют условие и исполняют один из нескольких условных выводов управления.

Обратим внимание на то, что диаграммы деятельности можно считать частным случаем диаграмм состояний.

Диаграмма последовательности (sequence diagram) предназначена для моделирования еще одной разновидности взаимодействия — взаимодействия во времени. С ее помощью отслеживается поведение взаимодействующих групп объектов. Диаграмма имеет два измерения.

- Слева направо, в виде линий жизни каждого объекта, участвующего во взаимодействии. Есть начальный момент времени — верхняя часть диаграммы, где указываются имена объектов.
- Сверху вниз, в виде вертикальных линий, представляющих линии жизни отдельных объектов. Линия жизни — время, в течение которого объект существует в системе.

Чтобы выделить активное состояние объекта, применяется специальное понятие — *фокус управления*, изображаемый как вытянутый прямоугольник, который становится продолжением линии жизни.

Сообщение — это законченный фрагмент информации, отправляемый одним объектом другому. Существуют четыре разновидности сообщений.

- Для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления.

- Для обозначения простого (не вложенного) потока управления.
- Для асинхронного сообщения между двумя объектами в некоторой процедурной последовательности.
- Для возврата из вызова процедуры.

3.2.3.8. Подходы к ведению анализа и проектирования

Структурная методология

Итак, комбинации структурных методов образуют структурные подходы. Можно выделить три группы структурных подходов на основе порядка построения модели [Калянов 1996].

- *Процедурно-ориентированные* подходы, в которых первично проектирование функциональных компонентов.
- Подходы, *ориентированные на данные*. Для таких подходов первичны входные и выходные данные, а функциональные (процедурные) компоненты вторичны.
- *Информационно-ориентированные* подходы. Эта группа близка к предыдущей, но отличается тем, что работа ведется с неиерархическими структурами данных.

О классах целевых систем

Можно выделить два класса целевых систем: информационные системы (управляемые данными) и системы реального времени (управляемые событиями). Информационные системы работают с большим объемом входных данных сложной структуры. Системы реального времени работают с малым количеством входных данных простой структуры. Как правило, для проектирования систем реального времени применяются подходы, базирующиеся на подходах для информационных систем с расширением их дополнительными диаграммными техниками.

Обычно подходы используют две основные группы средств моделирования [Вендров 2000].

- Диаграммы, иллюстрирующие функции, которые система должна выполнять, и связи между функциями. Например, диаграммы потоков данных и функционального моделирования.
- Диаграммы, моделирующие данные и их отношения. Например, диаграммы "сущность-связь".

О применении методов к российской специфике

Для анализа и проектирования экономических информационных систем в условиях российской действительности более подходят диаграммы потоков данных [Вендров 2000]. Диаграммы функционального моделирования неплохо работают только при описании хорошо стандартизированных процессов.

Перечислим основные подходы к ведению структурного анализа и проектирования и рассмотрим подробно некоторые из них.

- Подход Йордона и ДеМарко.
- Подход Гейна и Сарсона.
- Подход Константайна.
- Подход Джексона.
- Подход Варнье-Орра.
- Подход Мартина.
- Подход промышленной технологии DATARUN.
- Подход промышленного метода Oracle.

Подход Йордона и ДеМарко (Edward Yordon and Tom DeMarco) является процедурно-ориентированным и наиболее часто используемым (по статистике — в 36,5% случаев) [Калянов 1996]. Подход фокусирует внимание на потоках данных.

В нем интегрированы следующие средства:

- диаграммы потоков данных;
- словари данных, являющиеся каталогами всех элементов данных, присутствующих в диаграммах потоков данных;
- мини-спецификации обработки, описывающие процессы нижнего уровня. Фактически, мини-спецификации представляют собой алгоритмы описания задач, выполняемых процессами.

Подход Гейна и Сарсона (Chris Gane and Tom Sarson) очень близок к предыдущему. Статистика утверждает, что он применяется в 20,2% случаев [Калянов 1996]. Главной отличительной чертой подхода является наличие этапа моделирования данных, определяющего содержимое хранилищ данных в диаграммах потоков данных в третьей нормальной форме. Этот этап включает:

- построение списка элементов данных, располагающихся в каждом хранилище данных;
- анализ отношений между данными и построение соответствующей диаграммы связей между элементами данных;
- представление всей информации по модели в виде связанных нормализованных таблиц.

Подход Джексона (Michael Jackson) ориентирован на данные. Базовая процедура проектирования включает четыре этапа [Калянов 1996] [Кинг 1991].

- *Этап проектирования данных.* Построение системной сетевой диаграммы, демонстрирующей все хранилища, источники и стоки данных в программе. Представление каждой входной и выходной структуры данных древовидной структурной диаграммой.

- *Этап проектирования программ.* Формирование структуры программы комбинированием структур данных. Идентификация всех связей между компонентами структур данных.
- *Этап проектирования операций.* Построение списка операций, необходимых для продуцирования выходных структур данных из входных. Назначение операций компонентам структуры программы.
- *Этап проектирования текстов.* Трансляция построенной модели программы в текстовый вид с добавлением ряда логических условий для управления выполнением циклов и выбором данных.

Объектно-ориентированная методология

Перечислим основные подходы к ведению объектно-ориентированного анализа и проектирования и рассмотрим подробно некоторые из них.

- Подход на основе языка UML.
- Подход Шлеер-Меллора.
- Подход Гради Буча.
- Подход Джеймса Рамбо (Object Modelling Technique — OMT).
- Подход Ивара Якобсона (Object-Oriented Software Engineering — OOSE).

Подход на основе универсального языка моделирования UML включает следующие основные типы диаграмм:

- диаграммы вариантов использования;
- диаграммы классов;
- диаграммы состояний;
- три диаграммы поведения (две последние из них также называют диаграммами взаимодействия):
  - диаграммы деятельности;
  - диаграммы последовательности;
  - диаграммы кооперации;
- две диаграммы реализации:
  - диаграммы компонентов;
  - диаграммы развертывания.

Подход на основе языка UML состоит из четырех основных фаз разработки *(см. разд. 3.5.3.1)*, причем работа с диаграммами ведется в основном на второй и третьей фазах.

Во время второй фазы — фазы исследования — должна быть создана модель предметной области. Для этой цели наиболее естественно подходят следующие типы диаграмм:

- диаграммы вариантов использования;
- диаграммы классов;

- диаграммы деятельности.

На третьей фазе — фазе построения — продолжается итеративная работа с такими типами диаграмм, как:

- диаграммы классов;
- диаграммы деятельности.

К ним добавляются следующие типы диаграмм, которые определяют взаимодействие:

- диаграммы последовательности;
- диаграммы кооперации.

В случае сложного поведения системы разрабатывается еще одна группа диаграмм — диаграммы состояний.

Существует несколько рекомендаций, связанных с данным подходом:

- не следует стремиться к построению диаграмм всей системы;
- 80% проектов можно воплотить, используя 20% средств языка UML. Как правило, наиболее часто используют диаграммы классов и диаграммы деятельности.

Подход Шлеер-Меллора (Sally Shlaer and Stephen Mellor) использует три группы средств для создания модели предметной области.

- *Информационное моделирование* — для определения отношений между данными (информацией). При этом используется один тип диаграмм — диаграммы классов.
- *Моделирование состояний* — для определения, зависящего от времени поведения системы. Используются диаграммы состояний.
- *Моделирование процессов* — для определения функций, которые система должна выполнить. Используются.
  - диаграммы деятельности;
  - диаграммы последовательности.

Для анализа больших предметных областей используются диаграммы, по смыслу близкие к следующим диаграммам языка UML:

- диаграммы кооперации;
- диаграммы компонентов;
- диаграммы развертывания.

Подход предлагает механизм поддержки моделей состояний. Для этого вводятся четыре архитектурных класса.

- *Переход*, описывающий каждый переход для всех моделей состояний в программе.
- *Конечная модель состояний*, связывающая все экземпляры перехода, которые составляют одну модель состояний.

- *Активный экземпляр*. Это абстрактный класс, из которого все экземпляры, имеющие модель состояний, наследуют их текущее состояние.
- *Таймер*, обеспечивающий механизм функционирования таймеров на основе аппаратных средств, доступных для хранения следа времени.

3.2.4. Программирование (реализация)

3.2.4.1. Стиль программирования

Стиль — это отражение мышления.
А. Шопенгауэр

Важнейшая технологическая задача, возникающая в процессе программирования, — соответствие единому стилю программирования. Под *стилем программирования* обычно понимают [Тассел 1985] набор приемов или методов программирования, которые используют опытные программисты, чтобы получить правильные, эффективные, удобные для применения и легкочитаемые программы. Правила хорошего стиля — результат соглашения между опытными программистами. Обычно принципы хорошего стиля программирования являются результатом здравого смысла, исходящего из опыта. Это не набор обязательных правил, установленный раз и навсегда. Брайан Керниган (Brian Kernighan) и Роб Пайк (Rob Pike) уточняют, что код должен быть прост и понятен, т. е. обладать следующими свойствами [Керниган, Пайк 2001]:

- очевидная логика;
- естественные выражения;
- использование соглашений, принятых в языке разработки;
- осмысленные имена;
- аккуратное форматирование;
- развернутые комментарии;
- отсутствие хитрых трюков и необычных конструкций.

Правило стандартизации стиля заключается в том, что если существует более одного способа сделать что-либо и этот выбор произвольный, то следует остановиться на одном способе и всегда его придерживаться. Особое значение имеет единый стиль программирования в процессе работы над программным текстом в коллективных разработках. В большинстве крупных проектов существуют внутренние документы, определяющие стиль программирования команды разработчиков. Известным примером является документ по стилю программирования разработок GNU (**http://www.gnu.org/prep/standards_toc.html**). Применение таких документов приводит к использованию единого стиля, а, следовательно, к повышению читабельности и сопровождаемости программ, а во многих случаях и предотвращению большого класса ошибок. Обратим

внимание на то, что есть универсальные рекомендации, а есть рекомендации, связанные с конкретными языками.

Приведем некоторые рекомендации по стилю написания программ на языках C и C++, сгруппировав их в тематические классы. Большую часть рекомендаций с успехом можно применять и во многих других языках программирования. Многие рекомендации имеют обоснование, в некоторых случаях строгое, в других — слабое.

Структура файлов

Приведем рекомендации по именованию файлов.

- Заголовочные файлы должны иметь расширение h. Файлы с программой на языке C должны иметь расширение c, а с программой на языке C++ — cc (в операционной системе Unix) или cpp (в операционной системе Windows).
- Имена файлов должны отличаться уже первыми восемью символами. Потому что некоторые устаревшие, но, тем не менее, широко используемые операционные системы накладывают ограничения на длину имени файлов.
- Все файлы должны иметь различные имена, даже если они находятся в разных каталогах.

Опишем рекомендации по длине строк.

- Строки не должны быть длиннее 80 символов. Потому что многие текстовые редакторы и системы печати текстов программ ориентированы именно на это максимальное количество символов в строке. Исторически это объясняется текстовым режимом отображения символов на экране: 25 строк по 80 символов. Другое возможное объяснение — наследие холлеритовских перфокарт.
- Лучше всего, если длина строк программы значительно короче 80 символов.

Далее представлены рекомендации по организации файлов.

- Следует соблюдать следующий порядок в заголовочных файлах программы на языке C++:
  - комментарий к файлу, который оформляется как блочный комментарий. Комментарий может начинаться указанием на лицензию, которая определяет права собственности на данный файл. Комментарий обычно продолжается идентификационной строкой системы управления версиями файлов. Например, для утилиты SCCS *(см. разд. 5.6.1.1)* строка может выглядеть так:

    ```
    %Z%%M% %I% %E%
    ```

    и включать символы идентификации SCCS, имя файла, номер версии и дату последней модификации. Комментарий должен содержать краткое описание файла;

- защитная проверка, гарантирующая компиляцию данного файла один-единственный раз, вне зависимости от числа его включений в модуль компиляции. Она должна выглядеть так:

```
#if !defined(FILENAME_H)
#define FILENAME_H
```

- импортируемые интерфейсы. Как правило, они определяются с помощью `#include`-директив;
- объявления;
- завершающая защитную проверку директива условной компиляции `#endif`.

❐ Для файлов кода на языке C++ рекомендуется такая последовательность:

- комментарий к файлу;
- импортируемые интерфейсы;
- локальные объявления;
- экспортируемые определения.

Лексические соглашения

Приведем рекомендации по написанию идентификаторов.

❐ Не начинайте и не заканчивайте идентификаторы символом подчеркивания.

❐ Не используйте идентификаторы, состоящие только из прописных букв. Исключения составляют имена констант и макроопределений.

❐ Не следует в одной и той же программе использовать имена, различающиеся лишь написанием букв: строчной или прописной.

❐ Разделяйте части идентификаторов символом подчеркивания, а не написанием с прописной буквы очередной части идентификатора.

❐ Идентификаторы должны начинаться со строчной буквы.

Опишем соглашения об именовании.

❐ Смысл всех имен должен быть понятен при чтении программы. Имя переменной (в большинстве случаев) должно точно и полно раскрывать суть того, что она представляет.

❐ Длинные имена могут быть сокращены. Следует придерживаться либо удаления несущественных суффиксов, либо удаления всех гласных букв кроме начальной, либо использования нескольких первых букв слова. Рекомендуется использовать одни и те же методы сокращений, а также документировать все сокращения в специальных таблицах (например, в начале файла с программой).

- Имена функций должны быть активными, т. е. базироваться на активном глаголе, за которым следует существительное.
- Имена перечислимых типов могут иметь единый префикс.

В этом месте обратим внимание на то, что во многих случаях разумно использовать соглашения, традиционные для того или иного языка. Например, программистам на языке программирования C близки следующие правила.

- В качестве разделителя слов в идентификаторе следует использовать символ подчеркивания.
- Имена функций и переменных пишутся только строчными буквами.
- Имена макросов пишутся только прописными буквами.
- Имена, используемые в ограниченном контексте, могут быть очень короткими. Традиционно имена `i` и `j` используются для обозначения счетчиков, `p` и `q` — для указателей, `s` — для строковых, а `ch` — для литерных переменных. В данном случае ясность достигается краткостью.

Одним из наиболее продуманных наборов правил для составления имен переменных является "*венгерская нотация*". Имена переменных в ней составляются из трех частей.

- *Базовый тип* указывает на тип переменной. Обычно это не стандартный, а абстрактный тип.
- *Префиксы* описывают то, как используется данная переменная. Префиксы (в отличие от базовых типов) до некоторой степени стандартизованы. Префиксов может быть несколько, и пишутся они строчными буквами.
- *Спецификатор* — это содержательная часть имени, которая собственно и была бы именем, если бы не использовалась венгерская нотация. Спецификатор состоит из одного слова или последовательности слов, написанных без разделителя, причем каждое слово начинается с прописной буквы.

Приведем рекомендации по употреблению комментариев.

- Блочные комментарии следует применять для описания дополнительной информации о файле, классе, функции или переменной.
- Строковые комментарии следует использовать для коротких заметок в одну-две строки.

Языковые детали

Рекомендации по выравниванию приведены в следующем списке.

- Стандартная величина отступа равна четырем пробелам.
- Табуляционный отступ равен восьми пробелам.

Об отступах

Величина отступа сильно зависит от языка программирования. Существуют языки программирования, в которых вложенность блоков определяется отступом (например, Occam или Python). Со стандартным отступом такие программы становятся просто неправильными.

Приведем рекомендации по написанию выражений.

- Бинарные операторы должны быть отделены от операндов одним пробелом.
- Унарные операторы не следует отделять от операнда пробелом, если только этого не требуют правила языка.
- Используйте скобки, чтобы улучшить читаемость. Не следует добавлять пробелы к введенным скобкам.
- Старайтесь размещать выражение на одной строке.
- Если выражение переносится на следующую строку, то знак бинарной операции следует оставить на первой строке.

Приведем рекомендации по объявлению переменных.

- В каждом объявлении должна объявляться только одна переменная.
- Объявление переменной должно занимать одну строку.
- За типом переменной должны следовать пробел, имя переменной и комментарий с кратким описанием переменной.
- Если переменная является указателем, то символ `'*'` следует присоединять к типу.

Приведем рекомендации по написанию определения функций.

- Определение должно начинаться заголовком функции, за которым идут комментарий, описывающий назначение функции, и собственно тело функции.
- Открывающая и закрывающая фигурные скобки, определяющие тело функции, располагаются в первом столбце строк.
- Заголовок состоит из типа возвращаемого функцией значения, пробела, имени функции, за которым без пробела следует открывающая круглая скобка для списка параметров.
- Если функция статическая, то ключевое слово `static` должно быть вынесено в строку, предшествующую заголовку функции.
- Определение функции всегда начинается в самом левом столбце файла.
- Различные части функции следует отделять пустой строкой.

Приведем рекомендации по написанию условного оператора.

- По одному пробелу следует оставлять перед и после круглых скобок условия.
- Открывающая фигурная скобка должна находиться на той же строке, что и закрывающая круглая.

- Закрывающая фигурная скобка находится на том же уровне отступа, что и начало условного оператора.
- Если условие не помещается на одной строке, то его продолжение на следующей строке должно начинаться в том же столбце, что и первый символ выражения первой строки.
- Если часть `then` или `else` состоит из простого оператора, то открывающая и закрывающая фигурные скобки могут быть опущены.

Пример форматирования условного оператора приведен в листинге 3.1.

Листинг 3.1. Пример форматирования условного оператора

```
if ( some_result == NULL && some_calculation() ) {
    do_something();
    and_some_more();
}
else if ( some_value > main_result ) {
    do_something_else();
}
else {
    do_something_absolute_else();
}
```

Что касается рекомендаций по написанию оператора выбора, то основные соглашения в этом случае выглядят так же, как и для условного оператора. Пример выравнивания оператора выбора приведен в листинге 3.2.

Листинг 3.2. Пример форматирования оператора выбора

```
switch ( i ) {
    case 1:
        do_something();
        break;
    case 2: {
        do_something_else();
        and_some_more();
        break;
    }
    default:
        assert(false);
        break;
}
```

В основных соглашениях рекомендаций по написанию операторов цикла следует опять ориентироваться на условный оператор. Типичные примеры форматирования оператора цикла приведены в листинге 3.3.

Листинг 3.3. Пример форматирования оператора цикла

```
for ( i = some_initial_value();
      i >= some_other_value();
      i = some_new_value() ) {
    do_something();
    and_some_more();
}
do {
    do_something();
    and_some_more();
} while ( we_should_do() );
```

В рекомендациях по написанию оператора перехода самое важное (и практически единственное) правило заключается в том, что оператор перехода не следует использовать.

3.2.4.2. Стиль защитного программирования

Стиль — это человек.
Бюффон

Защитное программирование (defensive programming) — это такой стиль написания программ, при котором появляющиеся ошибки легко обнаруживаются и идентифицируются программистом [Тассел 1985].

Существуют три основных принципа защитного программирования.

- *Общее недоверие.* Для каждого модуля входные данные должны тщательно анализироваться в предположении, что они могут быть ошибочными.
- *Немедленное обнаружение.* Каждая ошибка должна быть выявлена как можно раньше, это упрощает установление ее причины.
- *Изолирование ошибки.* Ошибки в одном модуле должны быть изолированы так, чтобы не допустить их влияние на другие модули.

Принцип общего недоверия может быть поддержан различными способами. В стандартной библиотеке языка C существует макроопределение `assert(выражение)`, которое используется для включения в программу диагностических сообщений. Если выражение имеет значение нуль, то будет напечатано сообщение следующего вида:

```
Assertion failed: выражение, file имя_файла, line nnn
```

После этого будет вызвана библиотечная функция, которая завершит вычисления.

Теперь дадим несколько рекомендаций по защитному программированию [Тассел 1985].

- Делайте проверку области значений переменных.
- Выполняйте контроль правдоподобности значений переменных, которые не должны превышать некоторых констант или значений других переменных.
- Контролируйте итоги вычислений.
- Включайте автоматические проверки (например, контроль переполнения или потери точности).
- Проверяйте длину элементов информации.
- Проверяйте коды возврата функций.

О необходимости защитного программирования

К сожалению, излишняя самоуверенность или доверчивость по отношению к своим коллегам ведет к тому, что программисты не всегда пользуются защитным программированием. Следует заметить, что периодически все-таки случается то, что "не может быть никогда". Например, сбой в другом модуле программы может привести к такой ситуации в данном модуле, которая теоретически не должна была случиться.

3.2.4.3. Стиль грамотного программирования

> Глубоко ошибается тот, кто думает, что изделиями программистов являются программы, которые они пишут. Программист обязан изготовлять заслуживающие доверия решения и представлять их в форме убедительных доводов, а текст написанной программы является лишь сопроводительным материалом, к которому эти доказательства применимы.
>
> *Э. Дейкстра*

Грамотное программирование (literate programming) — это стиль написания программ, при котором программа рассматривается не как последовательность инструкций для компьютера, а как литературное произведение [Knuth 1992]. Таким образом, программист при написании программы формально описывает действия, которые он ожидает от компьютера, вставляя в соответствующие места код на языке программирования.

Концепция грамотного программирования была предложена Дональдом Кнутом в 1984 году в статье, в которой излагались новый способ документирования программ и реализующая этот способ система, названная "паутина" — WEB. Для реализации стиля система WEB предлагала одновременную работу над текстом программы и документацией разработчика к ней. Наибольшее распространение получила система CWEB, адаптированная к использованию C в качестве языка программирования. CWEB предполагает (**http://www.computerra.ru/offline/2001/387/7737/**), что программист структури-

рует свою работу, разбивая ее на "секции", которым могут соответствовать "модули", "подпрограммы или функции" или просто фрагменты кода, причем каждая секция состоит из трех частей.

- TeX-часть, содержащая документацию разработчика и разъясняющая содержимое секции.
- Промежуточная часть, содержащая специфические для секции макроопределения.
- C-часть, содержащая программный код.

Системой поддерживается синхронность нелинейного (состоящего из "переплетающихся" как паутина (web) кусков) кода и документации разработчика. Будучи использованной, WEB-система навязывает программисту элементы грамотности по Кнуту — концентрацию на изложении материала и стиле, начиная от декомпозиции программ на секции и заканчивая такими вещами, как выбор адекватных имен и использование понятных конструкций. Приверженцев стиля грамотного программирования можно встретить на сайте **http://www.literateprogramming.com/**, который содержит массу ссылок на инструментарий, документацию, публикации и другие сайты, посвященные этому стилю.

> ***О гипертекстовом грамотном программировании***
>
> В 1996 году Маркус Кнасмюллер (Markus Knasmuller) в статье "Reverse Literate Programming" предложил объединить стиль грамотного программирования и идею гипертекста. Если стиль грамотного программирования ориентирован на то, что сгенерированное описание будут читать последовательно (раздел за разделом), то результатом нового подхода является создание не только печатного документа, но и предоставление дополнительных возможностей при чтении кода с экрана. К ним можно отнести выборочный просмотр фрагментов кода, переход по ссылкам в графе потока управления и многие другие. Фактически программисту предоставляется возможность просматривать текст программы как гипертекстовый документ.

3.2.5. Тестирование и отладка

3.2.5.1. Введение в тестирование и отладку

Гораздо легче найти ошибку, нежели истину.

Гете

Тестирование — процесс выполнения программ с целью обнаружения факта наличия ошибок. Это классическое определение тестирования, принадлежащее Гленфорду Майерсу. Обратим внимание, в определении указан лишь один способ проведения тестирования. Существуют и методы ручного тестирования (например, инспекции и сквозные просмотры программ). Поэтому слова "...процесс выполнения программ..." разумно заменить на "...любая деятельность, выполняемая...".

Отладка — это процесс локализации и устранения ошибок.

Процессы тестирования и отладки схематически могут быть представлены так, как показано на рис. 3.4 [Липаев 1986]. Тестирование начинается с разработки множества тестов и их исполнения на основе одной из выбранных методик. После сравнения результатов с эталонными начинается либо диагностика проблемы (в случае расхождения результатов), либо оценка достаточности полноты тестирования. Подготовка дополнительных тестов потребуется при недостаточной полноте тестирования, невозможности локализовать проблему с помощью имеющихся тестов и необходимости выполнить контроль сделанного исправления (**http://www.testingfaqs.org/**).

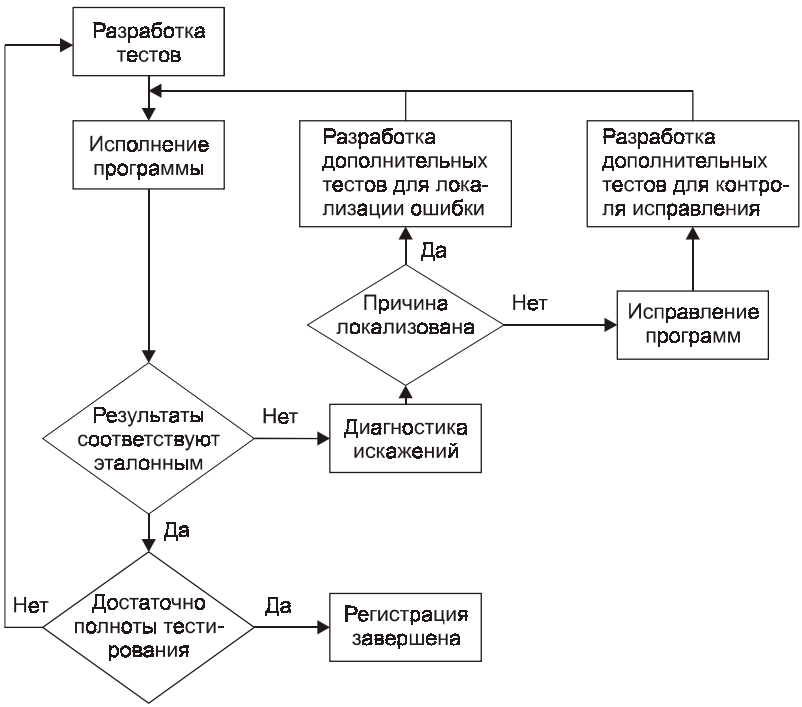


Рис. 3.4. Тестирование и отладка

3.2.5.2. Тестирование программных продуктов

Существует три вида лжи: ложь, наглая ложь и эталонные тесты.
Фольклор специалистов по разработке тестов

Сначала разберемся с типами и разновидностями тестирования. Существуют два основных метода тестирования.

- *Функциональное тестирование* программы, при котором программа рассматривается как объект, внутренняя структура которого неизвестна. Часто этот

метод известен как метод *черного ящика*. Это наиболее распространенный и популярный метод тестирования в промышленном масштабе.

- *Структурное тестирование* программы подразумевает знание исходного кода программы и полный доступ к нему. Этот метод также известен как метод *прозрачного (белого) ящика*.

Тестирование различают также по объему тестируемых единиц. Например, существует *тестирование процедур и функций*, *тестирование модулей*, тестирование *подсистем и систем*, а также *интеграционное тестирование*.

Тестирование с использованием компьютера обычно называют *динамическим тестированием*, причем оно разделяется на *автоматическое тестирование*, выполняемое инструментами (например, тестовым монитором) и *ручное тестирование*, выполняемое человеком. Ручное тестирование до сих пор широко применяется при тестировании графического пользовательского интерфейса. *Статическое тестирование* проводится без исполнения тестируемой программы на компьютере, путем ее анализа специалистами.

В зависимости от желаемой полноты тестирования выделяют ряд разновидностей, из которых упомянем следующие. *Детерминированное тестирование* контролирует каждую комбинацию исходных эталонных данных и соответствующую ей комбинацию результатов функционирования программ. На практике полное детерминированное тестирование обычно нереализуемо. *Случайное (стохастическое) тестирование* использует исходные тестовые данные, подготавливаемые случайным образом, как правило, с использованием статистического распределения. Так называемое *дымовое тестирование* (оно же "проверка сердцебиения") является очень непродолжительным (обычно не дольше, чем выкуривание одной или двух сигарет). Оно позволяет убедиться в работоспособности программы на ряде основных (впрочем, достаточно простых) тестов.

Вообще говоря, разновидностей тестирования очень и очень много, поскольку существует достаточно много критериев тестирования. Существует также несколько разновидностей, суть которых достаточно хорошо видна из названия: стрессовое тестирование, тестирование документации, тестирование интерфейсов, тестирование локализации, тестирование надежности, тестирование производительности и т. п.

Типичные ошибки

Мы уже говорили о необходимости защитного программирования, которое позволяет предупредить многие типичные ошибки. Обратим внимание на две категории наиболее распространенных ошибок программистов.

- Ошибки общего (несинтаксического) характера, остающиеся в программах после выполнения синтаксического контроля, такие как:
  - группа логических ошибок. Например, неполный учет возможных условий или неверное указание ветви алгоритма после проверки условия;

- ошибки в циклах. Например, неверные границы начала и конца;
- ошибки при работе с данными;
- ошибки ввода-вывода;
- ошибки в описании переменной. Например, отсутствие инициализации переменной;
- ошибки при работе с массивами;
- отсутствие начального обнуления элементов.

❐ Ошибки специального вида, особенно трудные для диагностирования, а именно:

- ошибки при написании параллельных программ. Например, ошибки в расстановках семафоров;
- ошибки, связанные с применением препроцессора;
- ошибки, связанные с неправильным результатом операций;
- исчезающие ошибки.

Тестовые данные

Для тестирования программ методом черного ящика [Канер, Фолк, Нгуен 2000] готовятся определенные группы тестов.

❐ Для тестирования *классов эквивалентностей*. Классы эквивалентностей позволяют вместо большого количества тестов использовать лишь их небольшое подмножество. Каждый тест представляет собой набор тестов, на которых программа ведет себя одинаково. Существует два основных типа классов эквивалентностей:

- класс корректных тестовых случаев, отражающих типичную "нормальную" ситуацию;
- класс тестов, содержащих ненормальную ситуацию, т. е. описывающих ситуацию, которой быть не должно.

❐ Для тестирования *граничных значений*.

❐ Для анализа *причинно-следственных связей*. Эти тесты применяются для программ, в которых взаимодействуют объекты.

❐ Для тестирования тех *утверждений*, которые приводятся в *документации*.

Для оценки результатов тестирования используются эталонные ("золотые") файлы, которые содержат ожидаемые результаты работы программы (эталонные результаты). Основные варианты получения эталонных результатов таковы:

❐ использование справочников;

❐ вычисление вручную;

❐ использование результатов, полученных при помощи другой программы, которой мы доверяем.

Тестовое покрытие — это набор тестов, покрывающих программу. Тестовое покрытие важно знать, чтобы определить участки кода, пропущенные при тестировании *(см. разд. 5.2.5.2)*.

Особый важный класс составляют тесты, *измеряющие производительность*. Существуют специальные группы и компании, которые разрабатывают такие тесты. Подробно оценка производительности вычислительных систем рассматривается *в разд. 6.2.5*.

Близкие по некоторому критерию тесты обычно объединяют в *тестовые сюиты*. На их основе готовятся *тестовые наборы*, применяемые для конкретных разновидностей тестирования.

Тестирование программ в процессе разработки

Поскольку в процессе разработки приходится тестировать еще не завершенную программу, все подходы к тестированию делятся на две группы.

- *Тестирование сверху вниз*. Применяется, если программа разрабатывается сверху вниз. В данном случае используются "заглушки" — фрагменты кода, имитирующие еще не написанные части программы.
- *Тестирование снизу вверх*. При этом, как правило, дополнительно должна быть создана небольшая программа — "драйвер", организующий взаимодействие уже написанных модулей.

Отслеживание ошибок

Проблема или ошибка может быть рассмотрена как некоторый объект в системе, который может находиться в одном из восьми состояний.

- Проблема или ошибка зарегистрирована.
- Выполнен первичный анализ (эксперт подтвердил факт наличия ошибки).
- Понятна причина, вызвавшая ошибку.
- Выполнен окончательный анализ проблемы или ошибки.
- Принято решение о начале работы над исправлением.
- Выполнено исправление ошибки.
- Исправление вошло в состав программного продукта.
- Подтверждено исправление ошибки.

Обратим внимание на то, что тест на обнаруженную проблему или ошибку должен быть обязательно помещен в тестовую базу данных. В результате получается регрессионная тестовая сюита. *Регрессионное тестирование* использует регрессионные тестовые сюиты, чтобы убедиться в том, что однажды исправленная ошибка не появилась вновь.

3.2.5.3. Отладка программных продуктов

Отладка программы — процесс творческий и плохо формализуемый. Тем не менее, основной идее отладки (базирующейся на анализе данных) можно придать вид следующего алгоритма:

1. Изучить уже доступные исходные и результирующие данные.
2. Сформулировать некоторую гипотезу, которая объясняет получение таких результирующих данных.
3. Подготовить новые исходные данные и провести эксперимент, который позволит доказать или опровергнуть гипотезу.

Обратим особое внимание на два достаточно простых случая отладки. Их анализ может быть достаточно легко выполнен при помощи пошагового отладчика.

- Если программа прекращает работу с порождениями "предсмертного дампа памяти" (core-файла), то практически любой отладчик позволит по этому дампу восстановить динамический стек вызовов процедур и, возможно, локализовать место проблемы.
- Если программа выдает некорректное сообщение, то можно получить динамический стек в отладчике, взяв за точку останова вызов функции выдачи сообщения. Стек может дать достаточное количество информации к размышлению.

> ***О первом выловленном баге***
>
> На компьютерном жаргоне ошибка в программе называется багом (от англ. bug — жучок). Происхождение этого термина связывают с мошкой, извлеченной в 1947 году из реле компьютера "Марк II". Грейс Мюррей Хоппер приклеила ее в журнал с примечанием "первый случай выловленного бага".

3.2.6. Ввод программы в действие

Ввод программы в действие существенно зависит от того, была ли это разработка для конкретного заказчика или она является продуктом, рассчитанным на широкого пользователя. Существует три основных способа доставки программы до пользователя.

- *Индивидуальная доставка* (как правило, разработка для конкретного заказчика).
- *Коробочная доставка.*
- *Доставка через Интернет.*

В последнем случае есть несколько основных разновидностей доставки.

- *Работа с программой* осуществляется через Интернет, как правило, *из среды интернет-браузера.* Пользователи имеют возможность получить по

сети исполняемый код (или его клиентскую часть) и начать работу с самой новой версией данного продукта. С подавляющим большинством таких приложений пользователи работают бесплатно. Однако ряд компаний развивают идею продажи сервисов работы с некоторым программным продуктом за определенную повременную оплату.

- Возможность *скачать из сети программу и установить ее на своих компьютерах*, подобно установке программы, поставляемой в коробке. Интернет неожиданно просто решает проблему распространения программного обеспечения и доставки его до пользователя. На смену десятилетию "коробочных продуктов" (1985—1995) пришли "продукты из Интернета". Причем в Интернете есть две группы программных продуктов:
  - свободно распространяемые программные продукты;
  - коммерческие программные продукты. На последние сначала можно получить временный регистрационный ключ (как правило, срок его действия около месяца), самостоятельно выяснить достоинства и недостатки продукта и принять решения о необходимости пользоваться им в дальнейшей работе. Если программный продукт подходит для пользователя, то он может получить постоянный регистрационный ключ после оплаты стоимости программного продукта.

Следует отметить, что в целом ряде случаев (например, работа с программой через Интернет) такое действие, как инсталляция (установка) программы, уходит в прошлое. Тем не менее, приведем следующие правила хорошего тона при подготовке инсталляции (**http://www.softshape.com/swrus/faq_swrus.html**).

- Следует, даже пользуясь инсталлятором, который упаковывает программу в один самораспаковывающийся файл, дополнительно сжимать его и распространять именно zip-архив. Не следует использовать другие архиваторы, даже если они сжимают лучше, поскольку их пользователь может не иметь.
- Следует всегда включать в архив текстовый файл README, который должен содержать подробную информацию о программе. Он должен быть доступен пользователю еще до начала инсталляции.
- Следует убедиться, что инсталлятор включает возможность автоматического удаления программы с диска. Программа без деинсталлятора создает массу лишних проблем своим пользователям.

3.2.7. Эксплуатация и сопровождение

Под *сопровождением* будем понимать все действия по повышению надежности программного продукта после завершения отладки и разработку усовершенствованных версий. Пожалуй, основным *принципом сопровождения* является закон Ласло Биледи (Laszlo Belady): используемый программный

продукт подвергается непрерывным изменениям для поддержания его экономической выгоды [Гласс, Нуазо 1983].

Перечислим четыре основные класса задач, решаемых на этапе сопровождения.

- *Адаптация*, как правило, заключающаяся в модификации функций.
- *Усовершенствование*, заключающееся обычно в добавлении новых функций.
- *Коррекция* или исправление ошибок.
- *Предупреждение* проблем, которые могут возникнуть в будущем.

Эти категории задач достаточно условны, и проблема может быть перенесена из одной категории в другую.

Для решения указанных задач выбирается один из следующих, наиболее подходящих типов сопровождения [Conger 1994], под которым будем понимать степень вмешательства в программу:

- *незначительные (локальные) изменения*;
- *реструктурирование кода* — повторная разработка небольшой части программы. При этом интерфейс этой части с остальной программой остается неизменным;
- *реинжиниринг* — перестройка существующего программного продукта. Обычно реинжиниринг применяется для приведения кода программы в соответствие с современными методологиями программирования (например, объектно-ориентированной);
- *программирование заново*.

На выбор типа сопровождения влияют два фактора.

- *Класс решаемой задачи.* Как правило, коррекция программы влечет незначительную степень вмешательства, а усовершенствование и адаптация — реструктурирование и реинжиниринг.
- *Стоимость сопровождения.* При высокой стоимости сопровождения рекомендуется ориентироваться на реструктурирование и реинжиниринг, а при низкой — на незначительные изменения.

К сожалению, считается, что сопровождением программ могут заниматься начинающие программисты, незнакомые с данной системой. Практика показывает, что сопровождение будет более эффективно и не приведет к нарушению концептуальной целостности продукта, если им будут заниматься опытные программисты.

Существует ряд методов, которые помогают добиться высокого качества сопровождения.

- *Выделение области связности изменений.* Следует четко представлять себе границы области, которая подвергается изменениям.

- *Самодокументируемость изменений.*
- *Пассивное и активное сопротивление изменениям.* За некоторое время до внесения изменения программист должен разослать письмо с описанием изменения и фрагментами изменяемого кода остальным участникам разработки. Программист сможет внести изменения только в следующих случаях:
  - если никто из коллег не выступит против (в случае пассивного сопротивления);
  - если все (вариант — или хотя бы один участник разработки) явно одобрят изменение (в случае активного сопротивления).

Проиллюстрируем процесс сопровождения на примере таких программ, как компиляторы. Поскольку они имеют ряд особенностей, характерных для их сопровождения [Safonov 1996], то перечислим эти особенности в порядке от более общих к более специфическим.

- Функционирование на нескольких аппаратных платформах.
- Наличие большого количества пользователей.
- Необходимость тщательного тестирования.
- Наличие устаревших алгоритмов.
- Наличие стандартов на языки программирования.

Источниками ("возбудителями") основных классов задач сопровождения могут быть:

- пользователи компилятора, находящие ошибки;
- разработчики взаимозависимых компонентов (например, отладчика);
- инженеры, сопровождающие компилятор (например, предлагая повысить эффективность компилятора);
- ученые, придумывающие новые алгоритмы и оптимизации.

Теперь рассмотрим, как четыре основных класса задач по сопровождению решаются применительно к компиляторам.

- Адаптация компиляторов предполагает их приспособление к изменившимся условиям работы системы. Все причины изменений могут быть сведены к трем основным.
  - *Архитектурные изменения.* Например, важнейшим типом адаптации является перенос компилятора на другую архитектуру.
  - *Изменения в операционной системе.* Например, операционная система может начать поддерживать многопоточное исполнение, начиная с некоторой версии.
  - *Изменения во взаимозависимых компонентах.* Например, разработка инкрементального редактора связей, позволяющего вносить

незначительные исправления в текст программы и продолжать исполнение без полной перекомпиляции.

- Усовершенствование компиляторов предполагает внесение в них изменений в интересах заказчика. Выделим несколько основных направлений усовершенствования.
  - *Повышение эффективности.* Например, оптимизация алгоритмов семантического анализа.
  - *Повышение надежности.* Информацию к размышлению здесь можно получить после обработки программы на языке C утилитой расширенного синтаксического анализа lint или утилитой поиска ошибок времени исполнения bcheck.
  - *Необходимость соответствия стандарту языка.* В случае изменения стандарта или появления комментариев к стандарту, компилятор должен все эти изменения учитывать.
  - *Необходимость соответствия популярному диалекту языка.* Как это ни печально, но иногда даже полностью соответствующий стандарту компилятор должен допускать (в специально задаваемом пользователем режиме) обработку конструкций, допускаемых другим популярным компилятором. Обычно это диктуется соображениями экономической выгоды.
- Если усовершенствование и адаптация могут быть рассмотрены для многих компиляторов без указания на язык программирования, то коррекция ошибок в большой степени ориентирована на компилятор с конкретного языка программирования. Исключение составляют специфические ошибки оптимизатора и кодогенератора. Лишь небольшое количество исправлений в части анализа может быть одинаково применено ко всему набору родственных компиляторов и, как правило, связано с неверным взаимодействием этой части с компонентами (например, ошибочная генерация информации отладчика).
- Предупреждение проблем может быть проиллюстрировано ставшей уже классической "проблемой 2000 года".

В большинстве случаев сопровождением программного продукта занимается компания, которая его разработала. Однако возможен вариант, когда сопровождение продукта передается другой компании.

3.2.8. Завершение эксплуатации

Ученью не один мы посвятили год,
потом других учить пришел и нам черед.
Какие ж выводы из этой всей науки?
Из праха мы пришли, нас ветер унесет.

Омар Хайям

Процесс завершения эксплуатации программного продукта несложен, но требует тщательной организационной подготовки. Обычно процесс начина-

ется с того, что пользователи заранее оповещаются о прекращении сопровождения программного продукта. С этого момента пользователи могут продолжать использовать продукт до тех пор, пока не наступит его моральное устаревание. Очень часто невозможность решить хотя бы одну из четырех задач сопровождения приводит к тому, что от продукта приходится отказаться.

3.3. Стандартные технологические процессы

Процессы жизненного цикла определяются международным стандартом ISO/IEC 12207 [ISO/IEC 12207:1995]. В данной книге русскоязычные формулировки стандартных процессов и действий, которые они включают, приведены согласно учебнику Александра Михайловича Вендрова [Вендров 2000]. Стандартные процессы разделяются на три группы: основные, вспомогательные и организационные процессы.

3.3.1. Основные процессы

> Некоторые удавы в процессе удушения так запутывались в собственных узлах, что потом приходилось тратить много времени на их распутывание.
>
> *Ф. Искандер. "Кролики и удавы"*

3.3.1.1. Приобретение

Процесс *приобретения* состоит из действий заказчика, приобретающего программное обеспечение. Процесс включает следующие действия:

- инициирование приобретения;
- подготовку заявочных предложений;
- подготовку и корректировку договора;
- надзор за деятельностью поставщика;
- приемку и завершение работ.

3.3.1.2. Поставка

Процесс *поставки* охватывает действия, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Процесс включает следующие действия:

- инициирование поставки;
- подготовку ответа на заявочные действия;
- подготовку договора;
- планирование;

- выполнение и контроль;
- проверку и оценку;
- поставку и завершение работ.

3.3.1.3. Разработка

Процесс *разработки* предусматривает действия, выполняемые разработчиком. Процесс охватывает работы по созданию программного обеспечения и его компонентов в соответствии с заданными требованиями. В процесс разработки включены следующие действия:

- подготовительная работа;
- анализ требований к системе;
- проектирование архитектуры системы;
- анализ требований к программному обеспечению;
- проектирование архитектуры программного обеспечения;
- детальное проектирование программного обеспечения;
- кодирование и тестирование программного обеспечения;
- интеграция программного обеспечения;
- квалификационное тестирование программного обеспечения;
- интеграция системы;
- квалификационное тестирование системы;
- установка программного обеспечения;
- приемка программного обеспечения.

3.3.1.4. Эксплуатация

Процесс *эксплуатации* охватывает действия оператора — организации, эксплуатирующей систему.

Процесс включает следующие действия:

- подготовительную работу;
- эксплуатационное тестирование;
- поддержку пользователей.

3.3.1.5. Сопровождение

Процесс *сопровождения* предусматривает действия, выполняемые службой сопровождения. Процесс включает следующие действия:

- подготовительную работу;
- анализ проблем и запросов на модификацию программного обеспечения;

- модификацию программного обеспечения;
- проверку и приемку;
- перенос программного обеспечения в другую среду;
- снятие программного обеспечения с эксплуатации.

3.3.2. Вспомогательные процессы

3.3.2.1. Документирование

Процесс *документирования* предусматривает формализованное описание информации, созданной в течение жизненного цикла программного обеспечения. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких как руководство, технические специалисты и пользователи системы. Процесс включает следующие действия:

- подготовительную работу;
- проектирование и разработку;
- выпуск документации;
- сопровождение документации.

3.3.2.2. Управление конфигурацией

Процесс *управления конфигурацией* предполагает применение административных и технических процедур на всем протяжении жизненного цикла программного обеспечения для:

- определения состояния компонентов программного обеспечения в системе;
- управления модификациями программного обеспечения;
- описания и подготовки отчетов о состоянии компонентов программного обеспечения и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов;
- управления хранением и поставкой программного обеспечения.

Процесс включает следующие действия:

- подготовительную работу;
- идентификацию конфигурации;
- контроль конфигурации;
- учет состояния конфигурации;
- оценку конфигурации;
- управление выпуском и поставкой.

3.3.2.3. Обеспечение качества

Процесс *обеспечения качества* предназначен для выполнения гарантий того, что программное обеспечение и процессы его жизненного цикла соответствуют заданным требованиям и утвержденным планам.

Процесс включает следующие действия:

- подготовительную работу;
- обеспечение качества продукта;
- обеспечение качества процесса;
- обеспечение прочих показателей качества системы.

3.3.2.4. Верификация

Процесс *верификации* состоит в определении того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, обусловленным предыдущими действиями.

Процесс включает два действия: подготовительную работу и собственно верификацию.

3.3.2.5. Аттестация

Процесс *аттестации* предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению.

Процесс включает два действия: подготовительную работу и собственно аттестацию.

3.3.2.6. Совместная оценка

Процесс *совместной оценки* предназначен для оценки состояния работ по проекту и программного обеспечения, создаваемого при выполнении данных работ.

Процесс включает три действия: подготовительную работу, оценку управления проектом и техническую оценку.

3.3.2.7. Аудит

Процесс *аудита* представляет собой определение соответствия программного продукта требованиям, планам и условиям договора. Особенность аудиторской проверки состоит в ее независимости.

Процесс включает два действия: подготовительную работу и аудит.

3.3.2.8. Разрешение проблем

Процесс *разрешения проблем* предусматривает анализ и решение проблем, которые обнаружены в ходе основных процессов, независимо от их происхождения или источника. Любая обнаруженная проблема должна быть идентифицирована, описана, проанализирована и разрешена.

Процесс включает два действия: подготовительную работу и собственно разрешение проблем.

3.3.3. Организационные процессы

3.3.3.1. Управление

Процесс *управления* состоит из действий, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона отвечает за управление проектом и управление действиями и задачами стандартных процессов.

Процесс включает следующие действия:

- инициирование и определение области управления;
- планирование;
- выполнение и контроль;
- проверку и оценку;
- завершение.

При инициировании процесса управления менеджер должен убедиться в наличии в его распоряжении необходимых ресурсов: человеческих, вычислительных и организационных.

3.3.3.2. Создание инфраструктуры

Процесс *создания инфраструктуры* охватывает выбор и поддержку технологий, стандартов и инструментальных средств, используемых для разработки, эксплуатации или сопровождения программного обеспечения. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам.

Процесс включает три действия: подготовительную работу, создание и сопровождение инфраструктуры.

3.3.3.3. Усовершенствование

Процесс *усовершенствования* определяет оценку, измерение, контроль и усовершенствование процессов жизненного цикла.

Процесс включает три действия: создание, оценку и усовершенствование процесса.

3.3.3.4. Обучение

Процесс *обучения* включает первоначальное обучение и последующее постоянное повышение квалификации персонала.

Процесс включает три действия: подготовительную работу, разработку учебных материалов и реализацию плана обучения.

3.3.4. Взаимосвязь между процессами

Вообще говоря, каждая компания может различным образом использовать стандартные процессы жизненного цикла программного обеспечения. Тем не менее, стандарт предлагает пять точек зрения на процессы.

- Заказчики и поставщики имеют контрактную (договорную) точку зрения. Процесс эксплуатации начинается тогда, когда заключаются контрактные отношения между заказчиком и поставщиком. Процесс поставки может привести (в зависимости от положений контракта) к началу процессов разработки, сопровождения или эксплуатации (рис. 3.5).
- Операторы и пользователи имеют эксплуатационную точку зрения.
- Разработчики системы и специалисты по сопровождению имеют инженерную точку зрения.
- Исполнители вспомогательных процессов имеют точку зрения поддержки.
- Менеджеры имеют управленческую точку зрения.

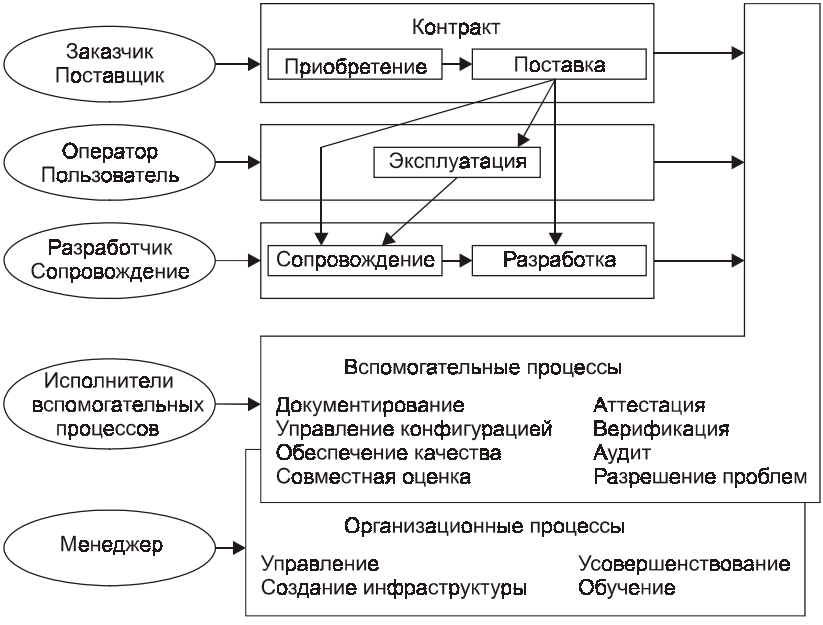


Рис. 3.5. Взаимосвязь между процессами

3.4. Основные стадии технологических подходов

Мы уже отмечали, что технологические стадии выделяются исходя из соображений разумного и рационального планирования и организации работ. Существует два основных варианта формирования временных рамок: фазы и стадии.

3.4.1. Фазы как крупные временные рамки

В этом случае названия фаз отражают крупные временные рамки. Пример такого деления присутствует в рациональном унифицированном процессе [Буч, Рамбо, Джекобсон 2000].

- Начало — определение бизнес-целей проекта.
- Исследование — разработка плана и архитектуры проекта.
- Построение — постепенное создание системы.
- Внедрение — поставка системы конечным пользователям.

3.4.2. Стадии как отражение классических процессов

Во втором варианте названия стадий отражают названия классических процессов (или их подмножества, или надмножества), большая часть времени которых проходит в данной стадии. Следующие девять стадий присутствуют практически во всех технологических подходах. Несколько упрощая суть, можно считать, что их содержание совпадает с рядом процессов и действий процесса разработки. По крайней мере, большая часть времени выполнения процесса или действия (действий) процесса будет приходиться на одноименную стадию:

- возникновение и исследование идеи;
- планирование;
- анализ требований;
- проектирование;
- программирование (реализация);
- тестирование и отладка;
- ввод в действие;
- эксплуатация и сопровождение;
- завершение эксплуатации.

3.4.3. Вариант подробного разбиения на стадии

Предложим руководителю проекта один из возможных вариантов подробного деления разработки на стадии. Он демонстрирует важность контрольных точек *(см. разд. 3.4.4)*. Этот вариант берет за основу стадии, отражающие классические процессы:

- стадия возникновения и исследования идеи;
- стадия планирования проекта;
- стадия анализа и проектирования;
- стадия реализации;
- стадия версии разработчика;
- стадия оценки жизнеспособности продукта;
- стадия альфа-версии;
- стадия бета-версии;
- стадия версии первой поставки пользователю.

Каждая стадия может быть разбита на этапы, отражающие основные события ее существования.

- Стадия возникновения и исследования идеи.
  - Этап 1 — Исследование идеи.
  - Этап 2 — Концентрация идеи в виде одностраничного описания.
  - Этап 3 — Принятие решения о начале работы над проектом.
- Стадия планирования проекта.
  - Этап 1 — Подготовка плана проекта.
  - Этап 2 — Техническое рецензирование проекта.
  - Этап 3 — Начало формирования команды разработчиков.
  - Этап 4 — Обзор планов инженеров.
  - Этап 5 — Исследование и поиск необходимых ресурсов.
  - Этап 6 — Достижение соглашения о выделении ресурсов.
  - Этап 7 — Утверждение проекта.
- Стадия анализа и проектирования.
  - Этап 1 — Анализ и проектирование продукта.
  - Этап 2 — Планирование разработки тестов и документации.
- Стадия реализации.
  - Этап 1 — Кодирование, разработка тестов и документации.
  - Этап 2 — Интеграция кода, тестов и документации.

- Стадия версии разработчика.
  - Этап 1 — Объединение кодов нескольких компонентов, составляющих проект.
  - Этап 2 — Стабилизация версии.
  - Этап 3 — Тестирование.
  - Этап 4 — Распространение версии.
- Стадия оценки жизнеспособности продукта.
  - Этап 1 — Оценка жизнеспособности продукта.
- Стадия альфа-версии.
  - Этап 1 — Объединение кодов нескольких компонентов, составляющих проект.
  - Этап 2 — Стабилизация версии.
  - Этап 3 — Формальное начало выпуска версии.
  - Этап 4 — Распространение версии.
  - Этап 5 — Продолжающаяся разработка.
  - Этап 6 — Формальное тестирование.
  - Этап 7 — Начало сопротивления изменениям.
- Стадия бета-версии.
  - Этап 1 — Объединение кодов нескольких компонентов, составляющих проект.
  - Этап 2 — Стабилизация версии.
  - Этап 3 — Создание электронного образа версии поставки.
  - Этап 4 — Формальное тестирование версии.
  - Этап 5 — Начало процесса помещения версии продукта на страницы в Интернете или на компакт-диски, дискеты или другие носители.
- Стадия версии первой поставки пользователю.
  - Этап 1 — Принятие решения о возможном переименовании бета-версии в версию первой поставки пользователю в случае отсутствия ошибок и переходе к этапу 6.
  - Этап 2 — Объединение кодов нескольких компонентов, составляющих проект.
  - Этап 3 — Стабилизация версии.
  - Этап 4 — Создание электронного образа версии поставки.
  - Этап 5 — Формальное тестирование версии.

- Этап 6 — Начало процесса помещения версии продукта на страницы в Интернете или на компакт-диски, дискеты или другие носители.
- Этап 7 — Завершающий анализ работы над версией ("разбор полетов").

К моменту завершения проекта

Статистики от стоматологии вывели закон "3 к 11". Они обнаружили, что пациенты распространяют информацию о зубных врачах следующим образом. Если зуб вылечен хорошо, то человек рассказывает об этом в среднем трем своим знакомым. Если плохо — тогда одиннадцати. Именно такова психология людей, в том числе и заказчиков. Именно это надо учитывать и добиваться полного удовлетворения заказчиков и пользователей, создавая о себе хорошую славу.

3.4.4. Контрольные точки

Важность *контрольных точек* заключается в том, что именно в них мы должны четко оценить, в какой мере достигнуты намеченные цели и критерии и не следует ли внести в процесс изменения прежде, чем двигаться дальше.

Уточним лишь, что традиционно понимается под различными типами версий и критериями, к ним предъявляемыми.

- *Версия разработчика* является ранней версией для внутреннего использования и тестирования (с учетом инсталлирования и лицензирования).
- На этапе *альфа-версии* продукт должен быть функционально полным, все фрагменты должны быть правильно интегрированы в систему. Обычно формулируется список специальных "альфа-критериев", которым продукт должен удовлетворять (например, должны быть исправлены все высокоприоритетные ошибки). Запрещается внесение новых свойств и изменение интерфейса. Готовы тестовые сюиты для генерального тестирования и документация. Как правило, программное обеспечение на этом этапе не передается для внешнего тестирования.
- На этапе *бета-версии* продукт должен удовлетворять специальным "бета-критериям" (например, должны быть исправлены все ошибки высокого и среднего приоритетов) и быть готовым к внешнему тестированию. Бета-версия содержит инсталляционную программу. Может быть полезным предоставление бета-тестерам информации об обнаруженных ошибках, в некоторых случаях это помогает найти причину ошибки.
- На этапе *версии первой поставки пользователю* продукт должен удовлетворять специальным "критериям первой поставки пользователю" (например, полное отсутствие существенных ошибок) и быть готовым к поставке пользователям.

3.5. Основные технологические подходы

Откуда мы пришли? Куда свой путь вершим?
В чем нашей жизни смысл? Он нам непостижим.
Как много чистых душ под колесом лазурным
сгорает в пепел, в прах, а где, скажите, дым?

Омар Хайям

3.5.1. Ранние технологические подходы

Ранние технологические подходы не используют явных технологий. Обычно их применяют только для очень маленьких проектов, как правило, завершающихся созданием демонстрационного прототипа. В качестве примера подхода, не использующего формализации, в данном разделе будет рассмотрен подход "кодирование и исправление".

3.5.1.1. Подход "кодирование и исправление"

Подход *"кодирование-исправление"* (code and fix) упрощенно может быть описан следующим образом. Разработчик начинает кодирование системы с самого первого дня, не занимаясь серьезным проектированием. Все ошибки обнаруживаются, как правило, к концу кодирования и требуют исправления через повторное кодирование.

Фактически каждый из программистов, так или иначе, применял этот подход. Практически все учебные программы пишутся в таком стиле. Следует заметить, что при использовании данного подхода затрачивается время лишь на кодирование, и заказчику легко демонстрировать прогресс в разработке в строках кода.

Этот подход может быть рекомендован к использованию в двух случаях.

- Для очень маленьких проектов, которые должны завершиться разработкой демонстрационного прототипа.
- Для демонстрации некоторой программной концепции.

3.5.2. Каскадные технологические подходы

Каскадные технологические подходы задают некоторую последовательность выполнения процессов, обычно изображаемую в виде каскада. Эти подходы также иногда называют подходами на основе модели водопада.

3.5.2.1. Каскадный подход

Каскадный подход (pure waterfall) считается "дедушкой" технологических подходов к ведению жизненного цикла. Фактически, его можно рассматривать как отправную точку для огромного количества других подходов. Сформировался каскадный подход в период с 1970 по 1985 годы. Специфика "чистого"

каскадного подхода такова, что переход к следующему процессу осуществляется только после того, как завершена работа с текущим процессом (см. рис. 3.1). Возвраты к уже пройденным процессам не предусмотрены.

Данный подход может быть рекомендован к применению в тех проектах, где в самом начале все требования могут быть сформулированы точно и полно. Например, в задачах вычислительного характера. Достаточно легко при таком технологическом подходе вести планирование работ и формирование бюджета.

О задачах вычислительного характера

Отметим, что сами вычислительные задачи очень часто решаются итерационными методами, т. е. именно путем многократного повторения одних и тех же действий.

3.5.2.2. Каскадно-возвратный подход

Основной недостаток каскадного подхода — отсутствие гибкости. Именно этот недостаток преодолевается *каскадно-возвратным подходом*, в котором разрешены возвраты к предыдущим стадиям и пересмотр или уточнение ранее принятых решений (рис. 3.6). Каскадно-возвратный подход отражает итерационный характер разработки программного обеспечения. Этот подход в значительной степени отражает реальный процесс создания программного обеспечения, в том числе и существенное запаздывание с достижением результата. На задержку оказывают существенное влияние корректировки при возвратах.

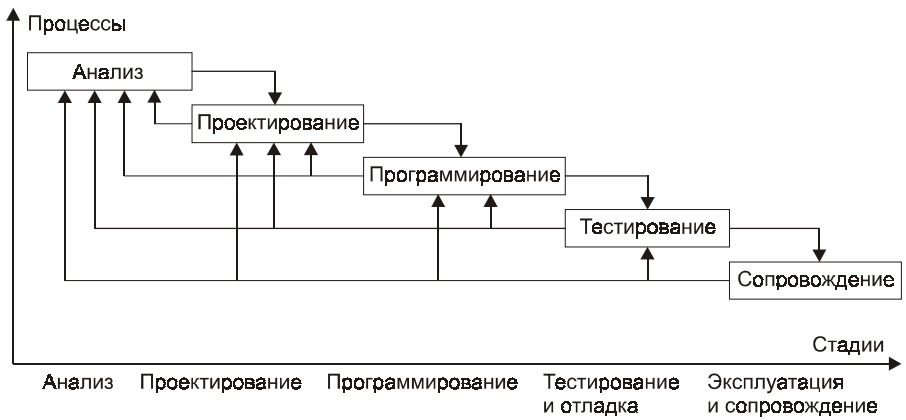


Рис. 3.6. Каскадно-возвратный технологический подход

3.5.2.3. Каскадно-итерационный подход

Каскадно-итерационный подход предусматривает последовательные итерации каждого процесса до тех пор, пока не будет достигнут желаемый результат (рис. 3.7). Каждая итерация является завершенным этапом, и ее итогом будет некоторый конкретный результат. Возможно, данный результат будет промежуточным, не реализующим всю ожидаемую функциональность.

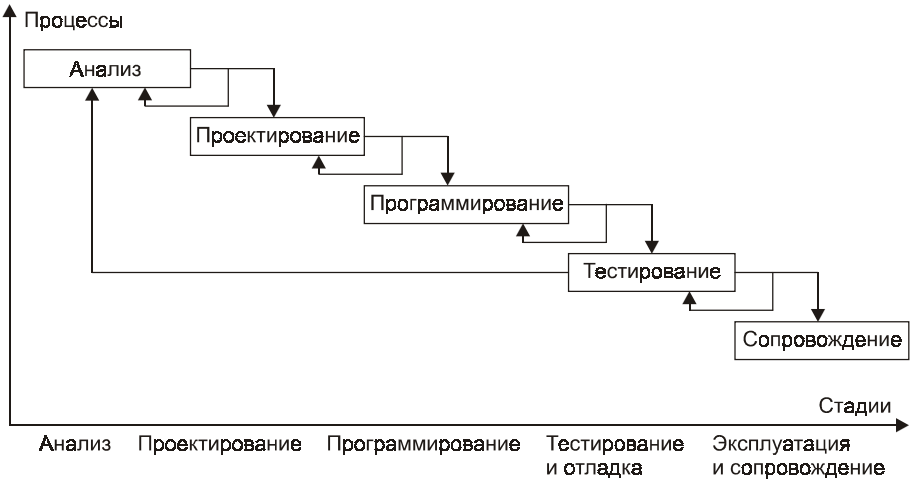


Рис. 3.7. Каскадно-итерационный технологический подход

3.5.2.4. Каскадный подход с перекрывающимися процессами

Классический каскадный подход позволяет выполнять каждый процесс отдельной команде. Достаточно разумным является использование команды на том же самом процессе в следующей разработке. *Каскадный подход с перекрывающимися процессами* (waterfall with overlapping) предполагает наличие специализированных команд, позволяющих до определенной степени сократить передаваемую документацию. Следующий процесс начинается до завершения текущего (рис. 3.8). Более того, несколько процессов могут выполняться параллельно.

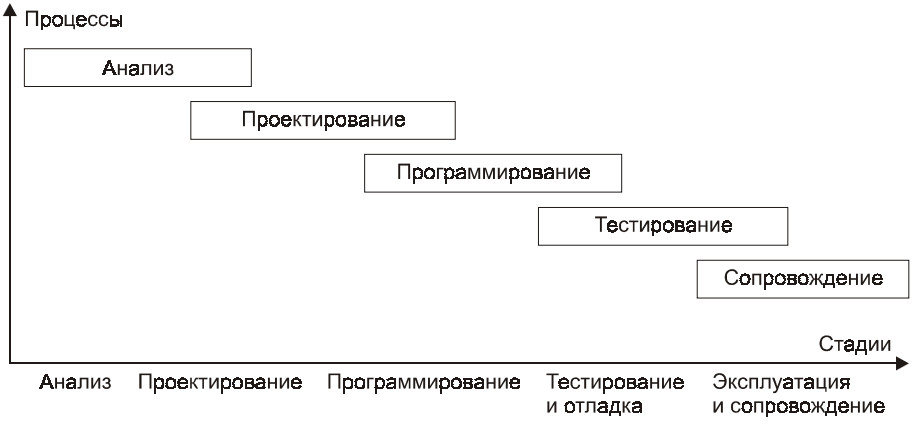


Рис. 3.8. Каскадный подход с перекрывающимися процессами

3.5.2.5. Каскадный подход с подпроцессами

Каскадный подход с подпроцессами (waterfall with subprocesses) очень близок подходу с перекрывающимися процессами. Особенность его в том, что с архитектурной точки зрения проект достаточно часто может быть разделен на подпроекты, которые могут разрабатываться индивидуально (рис. 3.9).

В данном подходе требуется дополнительная фаза тестирования подсистем до объединения их в единую систему. Следует особое внимание обращать на грамотное деление проекта на подпроекты, которое должно учесть все возможные зависимости между подсистемами.

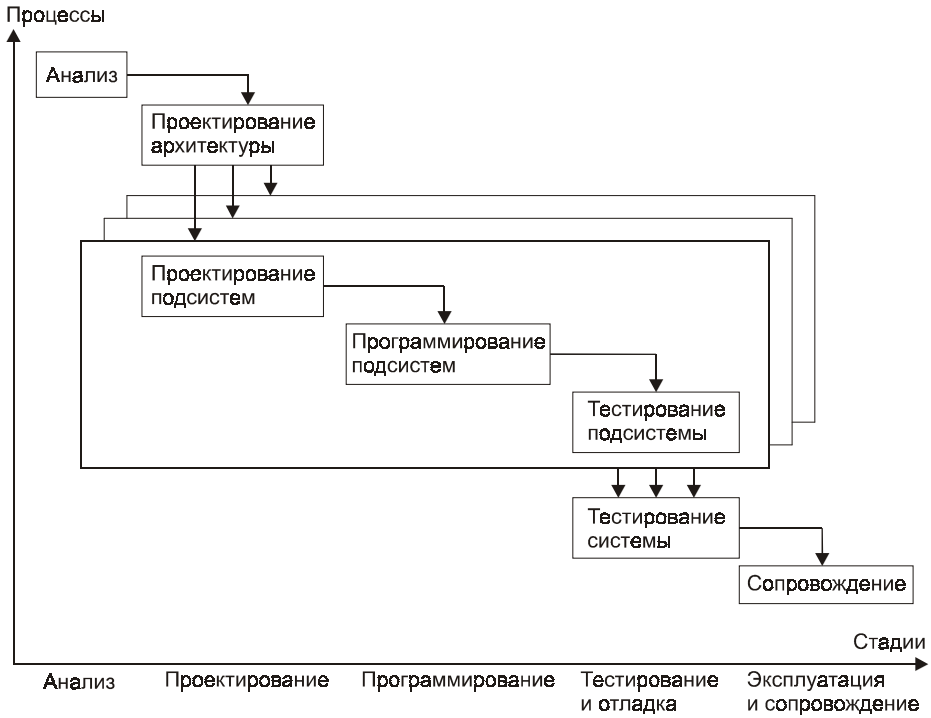


Рис. 3.9. Каскадный подход с подпроцессами

3.5.2.6. Спиральная модель

Спиральная модель (spiral model) была предложена Барри Боэмом (Barry Boem) в середине 80-х годов XX века с целью сокращения возможного риска разработки. Фактически, это была первая реакция на устаревание каскадной модели. Спиральная модель использует понятие прототипа, т. е. программы, реализующей частичную функциональность создаваемого программного продукта. Создание прототипов осуществляется за

несколько витков спирали, каждый из которых состоит из "анализа риска", "некоторого процесса" и "верификации" (рис. 3.10). Обращение к каждому процессу предваряет "анализ риска", причем если риск превышения сроков и стоимости проекта оказывается существенным, то разработка заканчивается. Это позволяет предотвратить более крупные денежные потери в будущем.

Особенность спиральной модели — в разработке итерациями. Причем каждый следующий итерационный прототип будет обладать большей функциональностью.

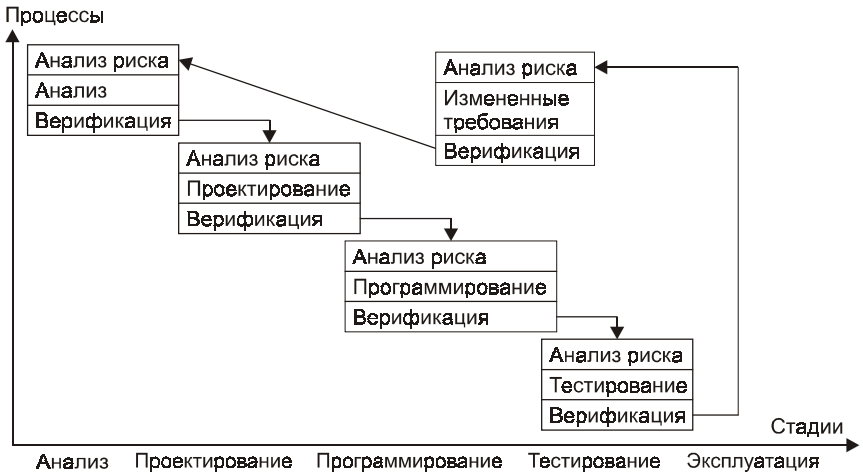


Рис. 3.10. Спиральная модель

3.5.3. Каркасные технологические подходы

Каркасные подходы представляют собой каркас для процессов и включают их огромное количество.

3.5.3.1. Рациональный унифицированный процесс

Рациональный унифицированный процесс (rational unified process) [Фаулер, Скотт 1999] [Буч, Рамбо, Джекобсон 2000] вобрал в себя лучшее из технологических подходов каскадной группы. В нем выделяются четыре основные фазы, рассмотренные ранее в *разд. 3.4.1.*

Именно в период прохождения этих фаз и функционируют процессы (например, анализ и проектирование), которые к тому же состоят из ряда последовательных итераций (рис. 3.11).

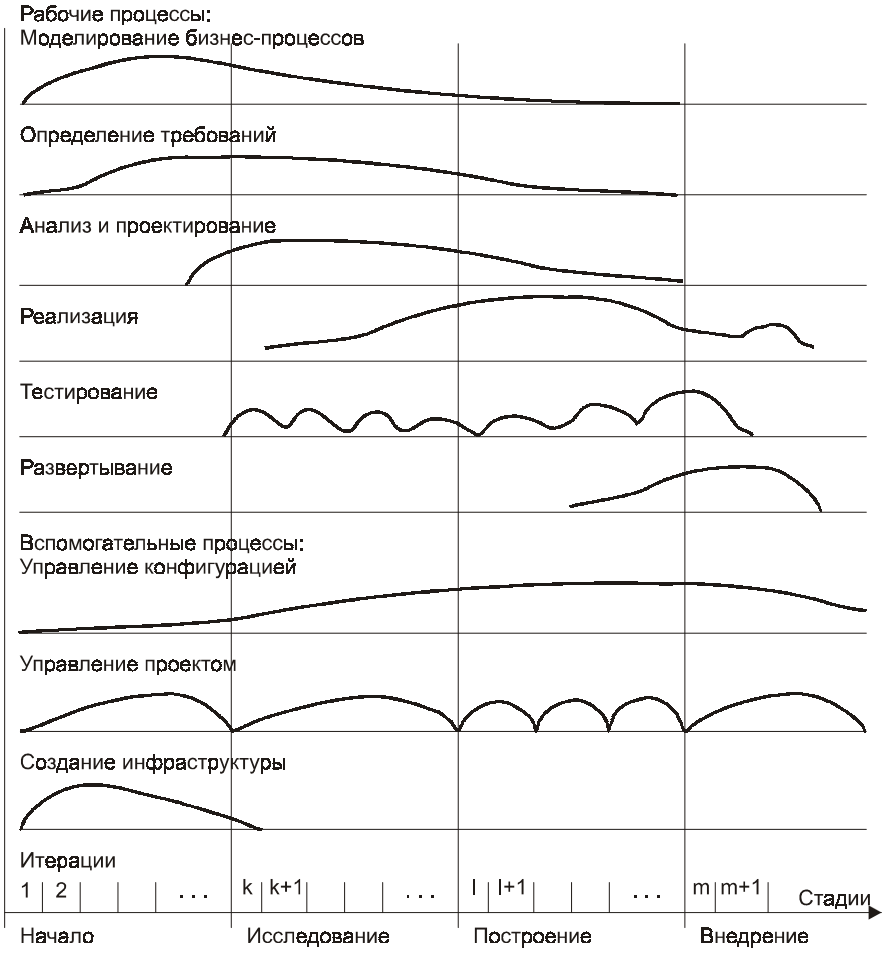


Рис. 3.11. Рациональный унифицированный процесс

Основные особенности данного подхода таковы:

- итеративность с присущей ей гибкостью;
- контроль качества, возможность выявить и устранить риски как можно раньше;
- предпочтение в первую очередь отдается моделям, а не бумажным документам;
- основное внимание уделяется раннему определению архитектуры;
- возможность конфигурирования, настройки и масштабирования.

3.5.3.2. Модель процессов Microsoft Solution Framework

Модель процессов Microsoft Solution Framework (MSF) (**http://www.microsoft.com/msf/**) представляет собой технологический подход, базирующийся на наборе моделей, правил и спецификаций, применяемых при создании и распространении программных продуктов, а также обеспечивающих более эффективное использование технологий для решения проблем бизнеса. Подход на основе MSF позволяет количественно выражать степень завершенности работы над проектом и адаптировать методы управления им в соответствии с изменяющимися потребностями современного бизнеса.

Модель процессов MSF возникла на основе используемого в компании Microsoft подхода к разработке программных приложений. В результате своего развития она объединила ряд наиболее эффективных принципов других известных моделей процессов, сформировав при этом единую базу для работы над проектами любых типов: ориентированных на фазы (phase-based), основанных на вехах/контрольных точках (milestone-driven) и итеративных (iterative).

Рамки (scope) — это сумма всех составляющих проекта, которые должны стать результатом работы над ним, а также все предоставляемые услуги, имеющие отношение к проекту. Рамки проекта определяют, что должно быть сделано для реализации единого видения. Они являются результатом компромисса между сформулированными целями и условиями реальности и отражают определение заказчиком приоритетов имеющихся требований к создаваемому решению. Частью процесса определения рамок проекта является вынесение менее важной функциональности из текущего проекта в планы на будущее. *Рамки решения (solution scope)* определяют функциональность решения и его возможности (включая те, что не относятся к программному обеспечению). Возможность (функциональность, составляющая, feature) — это требуемый или желаемый аспект программного или аппаратного обеспечения. Например, предварительный просмотр перед печатью может быть возможностью текстового процессора, шифрование почтовых сообщений — возможностью почтовой программы. Сопроводительные руководства пользователей, интерактивные файлы помощи, операционные руководства и обучение также могут быть составляющими (features) решения. *Рамки проекта (project scope)* определяют объем работ, который должен быть выполнен проектной группой для поставки заказчику каждого из элементов, определенного рамками решения.

Хорошо известна взаимозависимость между ресурсами проекта (людскими и финансовыми), его календарным графиком (временем) и реализуемыми возможностями (функциональностью). Эти три переменные образуют треугольник, показанный слева на рис. 3.12. После достижения равновесия в этом треугольнике изменение на любой его стороне для поддержания баланса требует модификаций на другой (двух других) сторонах и/или на изначально измененной стороне.

Другое весьма полезное средство для управления проектными компромиссами — матрица компромиссов проекта (project tradeoff matrix), показанная справа на рис. 3.12. Она отражает достигнутое на ранних этапах проекта соглашение между проектной группой и заказчиком о выборе приоритетов в возможных в будущем компромиссных решениях. В определенных случаях из этой расстановки приоритетов могут делаться исключения, но в целом следование ей облегчает достижение соглашений по спорным вопросам.

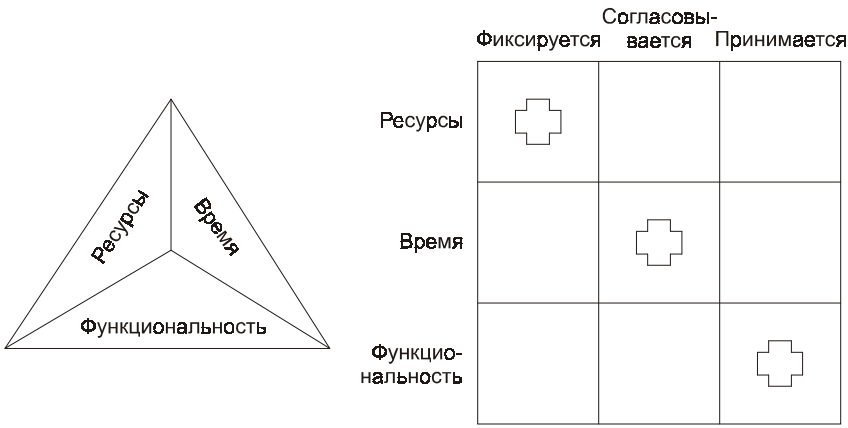


Рис. 3.12. Матрица компромиссов

Матрица компромиссов помогает обозначить проектное ограничение, воздействие на которое практически невозможно (колонка "Фиксируется"), фактор, являющийся в проекте приоритетным (колонка "Согласовывается"), и третий параметр, значение которого должно быть принято в соответствии с установленными значениями первых двух величин (колонка "Принимается"). Нельзя произвольно сокращать функциональность решения. Как проектная группа, так и заказчик должны тщательно проанализировать все имеющиеся в проекте ограничения и быть готовыми идти на обусловленные ими уступки. Для иллюстрации использования матрицы компромиссов рассмотрим следующее предложение (вместо пропущенных слов могут быть вставлены "ресурсы", "время" и "функциональность"): "зафиксировав <...>, мы согласовываем <...> и принимаем результирующие <...>". Матрице на рис. 3.12 соответствует следующее предложение: "зафиксировав ресурсы, мы согласовываем время и принимаем результирующую функциональность".

Тремя особенностями модели процессов MSF являются:

- *подход, основанный на фазах и вехах.* Вехи используются как опорные точки для планирования и мониторинга хода проекта. Главные вехи — это моменты жизненного цикла проекта, когда полученные на той или иной фазе результаты синхронизируются членами проектной группы друг с другом и с ожиданиями заказчика. В этот момент заказчиком,

заинтересованными сторонами и проектной группой производится формальный анализ достигнутого прогресса. Успешное прохождение главной вехи знаменует согласие проектной группы и заказчика продолжать далее работу над проектом;

- *итеративный подход.* Программный код, документация, дизайн, планы и другие рабочие материалы создаются, как правило, итеративными методами. MSF рекомендует начинать разработку решения с построения, тестирования и внедрения его базовой функциональности. Затем к решению добавляются все новые и новые возможности. Такая стратегия именуется стратегией версионирования;
- *интегрированный подход к созданию и внедрению решений.* Модель процессов MSF содержит весь жизненный цикл создания решения, включая его внедрение, вплоть до момента, когда решение начинает давать отдачу.

И, наконец, перечислим фазы и вехи модели процессов MSF.

- На фазе *выработки концепции* (envisioning phase) закладывается одна из фундаментальных основ успеха проекта — создание и сплочение проектной группы на основе выработки единого видения. Проектная группа должна четко представить себе, что она хочет сделать для заказчика и сформулировать свою цель таким образом, чтобы максимально мотивировать как заказчика, так и саму проектную команду. Выработка высокоуровневого взгляда на цели и условия проекта может рассматриваться как ранняя форма планирования, она подготавливает почву для процессов создания детальных планов, которые будут осуществлены непосредственно во время фазы планирования.
- На фазе *планирования* (planning) производится основная работа по составлению планов проекта. Она включает в себя подготовку проектной группой функциональной спецификации, разработку дизайнов, подготовку рабочих планов, оценку проектных затрат и сроков разработки различных составляющих проекта.
- На фазе *разработки* (developing) проектная группа фокусируется на создании компонент решения (включая как документацию, так и программный код). Однако некоторая часть этой работы может продолжаться также на фазе стабилизации, если такая необходимость выявлена в процессе тестирования. Данная фаза также включает в себя разработку инфраструктуры.
- Во время фазы *стабилизации* (stabilizing) производится тестирование разработанного решения. При этом внимание фокусируется на его эксплуатации в реалистичной модели производственной среды. Проектная группа занимается определением приоритета ошибок и их устранением, а также подготовкой решения к выпуску.
- Во время фазы *внедрения* (deploying) проектная группа внедряет технологии и компоненты решения, стабилизирует внедренное решение,

передает работу персоналу поддержки и сопровождения и получает со стороны заказчика окончательное одобрение результатов проекта. По завершению внедрения проектная группа производит анализ выполненной работы и удовлетворенности заказчика.

3.5.4. Генетические технологические подходы

Название этой группы подходов дано под впечатлением от статьи Игоря Васильевича Поттосина [Поттосин 1997], в которой термин "генетический" связывается с происхождением программы и дисциплиной ее создания.

3.5.4.1. Синтезирующее программирование

Синтезирующее программирование предполагает синтез программы по ее спецификации. В отличие от программы, которая написана на алгоритмическом языке и предназначена для исполнения на вычислительной машине после трансляции в исполняемый код, документ на языке спецификаций является лишь базисом для последующей реализации. Для получения этой реализации необходимо решить следующие основные задачи.

- Доопределить детали, которые нельзя выразить при помощи языка спецификации, но необходимые для получения исполняемого кода.
- Выбрать язык реализации и аппаратно-программную платформу для реализации.
- Зафиксировать отображение понятий языка спецификаций на язык реализации и аппаратно-программную платформу.
- Осуществить трансформацию представления (из спецификации в исполняемую программу на языке реализации).
- Отладить и протестировать исполняемую программу.

Подчеркнем, что если спецификация не содержит полного описания способа решения задачи (например, для компилятора не описан метод контроля семантической правильности входного текста), то автоматическая генерация программы невозможна.

Автоматическая генерация программ по спецификациям может быть выполнена для многих языков спецификаций, среди которых особо выделим SDL, ASN.1, LOTOS, Estelle и UML.

3.5.4.2. Сборочное (расширяемое) программирование

Сборочное программирование предполагает, что программа собирается путем переиспользования уже известных фрагментов (рис. 3.13).

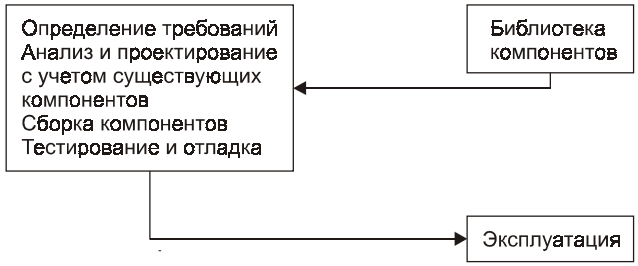


Рис. 3.13. Сборочное программирование

Сборка может осуществляться вручную или быть задана на некотором языке сборки, или извлечена полуавтоматическим образом из спецификации задачи. В статье [Цейтин 1990] изложены основные направления для создания техники сборочного программирования.

- Выработка *стиля программирования*, соответствующего принятым принципам модульности.
- Повышение *эффективности межмодульных интерфейсов*; важность аппаратной поддержки модульности.
- Ведение *большой базы программных модулей*, решение проблемы идентификации модулей и проверки пригодности по описанию интерфейса. Модули должны стать "программными кирпичиками", из которых строится программа.

Сборочное программирование тесно связано с методом повторного использования кода, причем как исходного, так и бинарного. Выделяют несколько разновидностей технологических подходов сборочного программирования, которые в значительной степени определяются базисной методологией.

- *Модульное сборочное программирование.* Этот подход был исторически первым и базировался на процедурах и функциях методологии структурного императивного программирования.
- *Объектно-ориентированное сборочное программирование.* Подход базируется на методологии объектно-ориентированного программирования и предполагает распространение библиотек классов в виде исходного кода или упаковку классов в динамически компонуемую библиотеку.
- *Компонентное сборочное программирование.* Основные идеи подхода — распространение классов в бинарном виде и предоставление доступа к методам класса через строго определенные интерфейсы, что позволяет снять проблему несовместимости компиляторов и обеспечивает смену версий классов без перекомпиляции использующих их приложений. Существуют конкретные технологические подходы, поддерживающие компонентное сборочное программирование: COM (DCOM, COM+), CORBA, .NET.

- *Аспектно-ориентированное сборочное программирование.* Концепция компонента в этом случае дополняется концепцией *аспекта* — варианта реализации критичных по эффективности процедур. Аспектно-ориентированное сборочное программирование заключается в сборке полнофункциональных приложений из многоаспектных компонентов, инкапсулирующих различные варианты реализации.

3.5.4.3. Конкретизирующее программирование

Конкретизирующее программирование предполагает, что частные, специальные программы извлекаются из универсальной.

Наиболее известная технология конкретизирующего программирования — это подход с применением паттернов проектирования [Гамма, Хелм, Джонсон, Влиссидес 2001]. *Паттерн (шаблон) проектирования* (design pattern) — описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Проектировщик, знакомый с паттернами, может сразу применять их к решению новой задачи, конкретизируя их. Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений.

Паттерн состоит из четырех основных элементов:

- *имени*, однозначно описывающего проблему проектирования;
- *задачи* — описания того, когда следует применять паттерн для конкретизации;
- *решения* — абстрактного описания элементов дизайна и отношений между ними;
- *результатов* — следствий применения паттерна.

Дополнительно к паттернам существуют *каркасы* (framework) — наборы взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ. Каркас диктует определенную архитектуру приложения, в нем аккумулированы проектные решения, общие для проектной области. Например, существуют каркасы, которые используются для разработки компиляторов.

3.5.5. Подходы на основе формальных преобразований

Эта группа подходов содержит максимально формальные требования к процессу создания программного обеспечения.

3.5.5.1. Технология стерильного цеха

Основные идеи *технологии стерильного цеха* (cleanroom process model) были предложены Харланом Миллзом (Harlan Mills) в середине 80-х годов

XX века. Технология складывается из следующих частей (рис. 3.14) [Linger 1994].

- Разработка функциональных и пользовательских спецификаций.
- Инкрементальное планирование разработки.
- Формальная верификация.
- Статистическое тестирование.

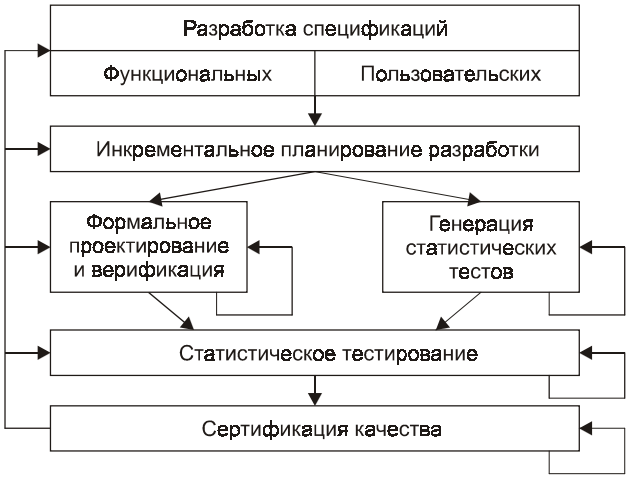


Рис. 3.14. Технология стерильного цеха

Процесс проектирования связан с представлением программы как функции, в виде так называемых "ящиков".

- *Черного ящика* с фиксированными аргументами (стимулами) и результатами (ответами).
- *Ящика с состоянием*, в котором выделяется внутреннее состояние.
- *Прозрачного (белого) ящика*, представляющего реализацию в виде совокупности функций при пошаговом уточнении.

> ***Замечание о методе ящиков***
>
> Обратим внимание на то, что понятия "черный" и "прозрачный" ящик уже встречались нам при рассмотрении тестирования программных продуктов. Метод ящиков удобен, когда надо работать с функциями либо с закрытой, либо с открытой реализацией.

Использование ящиков определяют следующие три принципа.

- Все определенные при проектировании данные скрыты (инкапсулированы) в ящиках.

- Все процессы определены как использующие ящики последовательно или параллельно.
- Каждый ящик занимает определенное место в системной иерархии.

Черный ящик представляет собой точную спецификацию внешнего, видимого с пользовательской точки зрения поведения. Ящик получает стимулы `S` от пользователя и выдает ответ `R`. Каждый ответ черного ящика определяется его текущей историей стимулов `SH` как следующая функция:

```
(S, SH) -> (R)
```

На основании одних и тех же стимулов мы можем получить разные ответы, базирующиеся на истории использования. Рассмотрим калькулятор с двумя историями стимулов:

- `Clear 3 1 4`
- `Clear 3 1 4 +`

Если следующим стимулом будет число 1, то на основании первой истории калькулятор выдаст ответ 3141, а на основании второй — 1. Определять поведение черных ящиков очень удобно с помощью табличных спецификаций.

Ящик с состояниями можно получить из черного ящика выделением элементов истории стимулов, которые сохраняют состояние (инварианты состояний) в процессе выполнения действий в черном ящике. Функция преобразования ящика с состояниями выглядит так:

```
(S, OS) -> (R, NS)
```

Здесь `OS` — старое состояние, а `NS` — новое. Хотя с пользовательской точки зрения поведения черного ящика и ящика с состоянием выглядят одинаково, истории стимулов заменяются ссылками на старое состояние и новое состояние, требуемое преобразованием.

Прозрачный ящик можно получить из ящика с состояниями, определяя процедуру, выполняющую требуемое преобразование. Таким образом, прозрачный ящик — это просто программа, реализующая соответствующий ящик с состоянием.

```
(S, OS) -> (R, NS) by procedure
```

Проектирование процедуры прозрачного ящика выглядит как последовательное определение функций назначения и преобразование их в структуру управления и новые функции назначения. Пример проектирования процедуры прозрачного ящика [Linger 1994] приведен на рис. 3.15.

Обратим внимание на то, что в данной технологии практически отсутствует такой процесс, как отладка. Его заменяет процесс формальной верификации. Для каждой управляющей структуры проверяется соответствующее условие корректности.

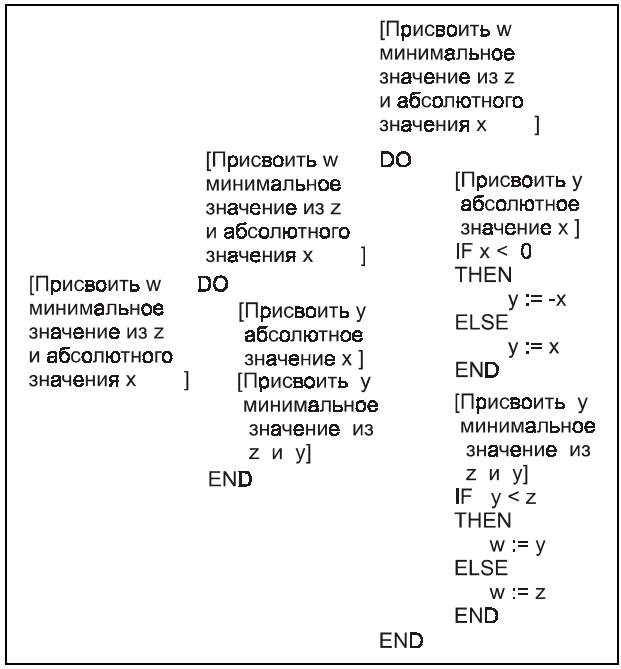


Рис. 3.15. Пример проектирования процедуры прозрачного ящика

Технология стерильного цеха предполагает бригадную работу, т. е. проектирование, уточнение, инспекция и подготовка тестов ведется разными людьми.

3.5.5.2. Формальные генетические подходы

На сегодняшний день сложились методы программирования, обладающие свойством доказательности и не теряющие это точное, накопленное знание [Поттосин 1997]. Три таких метода соответствуют уже исследованным генетическим подходам, но с учетом формальных, математических спецификаций.

- *Формальное синтезирующее программирование* использует математическую спецификацию — совокупность логических формул. Существуют две разновидности синтезирующего программирования:
  - *логическое*, в котором программа извлекается как конструктивное доказательство из спецификации, понимаемой как теорема;
  - *трансформационное*, в котором спецификация рассматривается как уравнение относительно программы и символическими преобразованиями превращается в программу.
- *Формальное сборочное программирование* использует спецификацию как композицию уже известных фрагментов.
- *Формальное конкретизирующее программирование* использует такие подходы, как смешанные вычисления и конкретизацию по аннотациям.

Одной из наиболее интересных современных работ в области формальных генетических подходов является B-технология [Abrial 1996]. На ее основе была осуществлена разработка системы управления парижским метрополитеном "METEOR".

3.5.6. Группа ранних подходов быстрой разработки

Развитием и одновременно альтернативой каскадных подходов является группа подходов быстрой разработки. Все эти подходы объединяют следующие основные черты:

- итерационную разработку прототипа;
- тесное взаимодействие с заказчиком.

3.5.6.1. Эволюционное прототипирование

Первый прототип при *эволюционном прототипировании* (evolutionary prototyping) обычно включает создание развитого пользовательского интерфейса. Он может быть сразу же продемонстрирован заказчику для получения от него отзывов и возможных корректив. Основное начальное внимание уделяется стороне системы, обращенной к пользователю. Далее, до тех пор, пока пользователь не сочтет программный продукт законченным, в него вносится необходимая функциональность (рис. 3.16).

Эволюционное прототипирование разумно применять в тех случаях, когда заказчик не может четко сформулировать свои требования к программному продукту на начальных этапах разработки или заказчик знает, что требования могут кардинально измениться.

Существенным недостатком данного подхода является невозможность определить продолжительность и стоимость проекта. Неочевидным является количество итераций, по истечении которых пользователь сочтет программный продукт законченным.

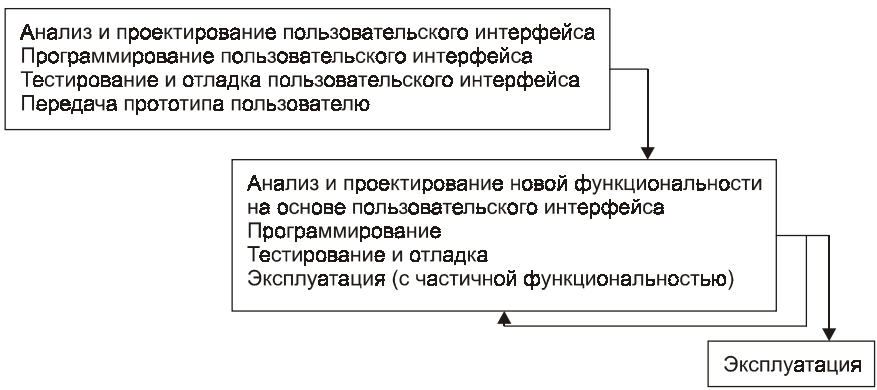


Рис. 3.16. Эволюционное прототипирование

Разновидностью данного подхода является эволюционное переписывание с выбрасыванием прототипа. В этом случае следующий прототип переписывается заново с учетом опыта разработки предыдущего варианта, как правило, неудачного.

3.5.6.2. Итеративная разработка

Первый прототип *итеративной разработки* (iterative delivery) уже должен включать завершенное ядро системы. Таким образом, в нем уже сосредоточена большая часть функциональности. Очередные итерации должны помочь пользователю определиться с доводкой пользовательского интерфейса, генерируемых системой отчетов и других выходных данных (рис. 3.17).

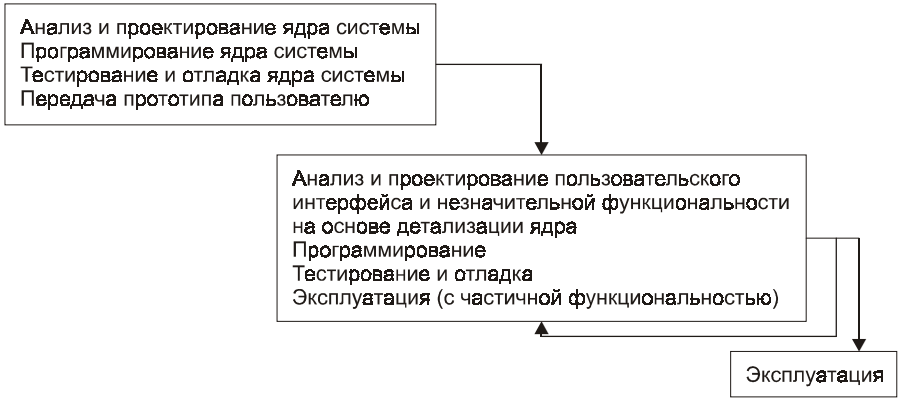


Рис. 3.17. Итеративная разработка

Допускается добавление незначительной функциональности, обычно не затрагивающей ядро системы.

Подчеркнем еще раз различие между двумя исследованными методами быстрой разработки. Итеративная разработка начинается с создания ядра системы и далее детализирует его. В отличие от этого, эволюционное прототипирование ориентировано на начальную разработку пользовательского интерфейса и добавления функциональности на его основе.

3.5.6.3. Постадийная разработка

Постадийная разработка (staged delivery) предназначена решить недостаток двух предыдущих подходов — невозможность определения сроков завершения проекта. Начиная разработку, мы достаточно хорошо знаем, что будет собой представлять создаваемый программный продукт. Основная задача постадийной разработки — предоставить заказчику работающую систему как можно раньше. Далее заказчик сможет добавлять новую функциональность

и получать очередную версию системы. Однако каждая из версий, получаемых по завершении стадий, является работающей.

Следует заметить, что данный подход требует тщательного и серьезного тестирования очередной системы в конце каждой стадии перед передачей ее пользователю.

3.5.7. Адаптивные технологические подходы

Адаптивные технологические подходы были задуманы как подходы, поддерживающие изменения. Они только выигрывают от изменений, даже когда изменения происходят в них самих. Данные подходы ориентированы на человека, а не на процесс. В них необходимо учитывать в работе природные качества человеческой натуры, а не действовать им наперекор (**http://www.martinfowler.com**).

3.5.7.1. Экстремальное программирование

Наиболее концентрированно идеи быстрой разработки программ оказались выражены в подходе *экстремального программирования* (extreme programming) (**http://www.extremeprogramming.org**). Две основные черты, присущие быстрым разработкам, являются базовыми и в этом подходе. Методы, объединенные в данном подходе, не являются принципиально новыми. Однако именно их рациональное объединение и совокупное использование дает существенные результаты и успешно выполненные проекты [Бек 2000]. Наибольшую пользу подход экстремального программирования может принести в разработке небольших систем, требования к которым четко не определены и вполне могут измениться.

Подход начинается с того, что проводится анализ назначения системы и определения первоочередной функциональности. В результате составляется список историй — возможных применений системы. Каждая история должна быть ориентирована на определенные задачи бизнеса, которые можно оценить с помощью количественных показателей. Наконец, ценность истории определяется материальными и временными затратами на ее создание командой разработчиков.

Заказчик выбирает истории для очередной итерации, основываясь на их значимости для проекта и ценности. Для первой версии системы заказчик определяет небольшое количество логически связанных наиболее важных историй. Для каждой следующей версии выбираются наиболее важные истории из числа оставшихся (рис. 3.18).

Одним из существенных методов данного подхода является функциональное тестирование. Существуют две особенности процесса тестирования.

- Программисты сами пишут тесты для тестирования программы.
- Эти тесты пишутся до начала кодирования.

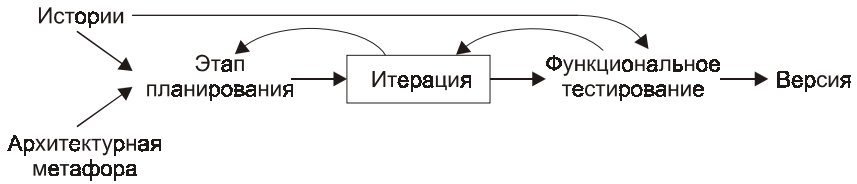


Рис. 3.18. Работа над проектом на основе экстремального программирования

Уверенность в нормальной работе каждого отдельного теста постепенно формирует у разработчиков уверенность в нормальной работе системы в целом.

Цель каждой итерации (рис. 3.19) — включить в версию несколько новых историй. На собрании по планированию итерации определяется, какие именно истории будут реализованы и каким образом это будет сделано командой разработчиков.

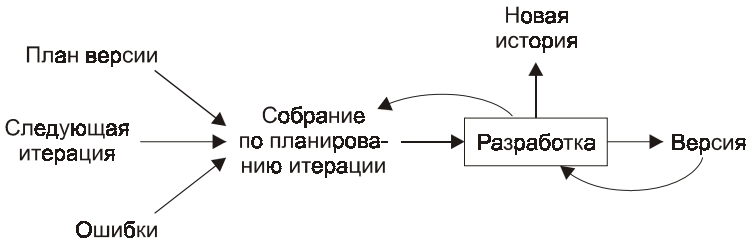


Рис. 3.19. Итерация

Коллективное владение кодом в процессе разработки (рис. 3.20) означает возможность для каждого программиста в любое время усовершенствовать любую часть кода в системе, если он сочтет это необходимым. Программист берет на себя ответственность за реализацию определенных задач. В случае возникновения вопросов о разрабатываемой задаче, может быть проведено короткое (15-минутное) собрание с присутствием заказчика.

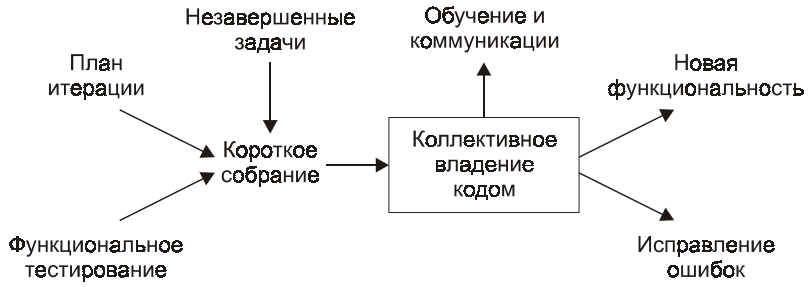


Рис. 3.20. Разработка

Для того чтобы выполнить задачу, ответственный за нее программист должен найти себе партнера (рис. 3.21). Окончательный код всегда пишется двумя программистами на одной рабочей станции.

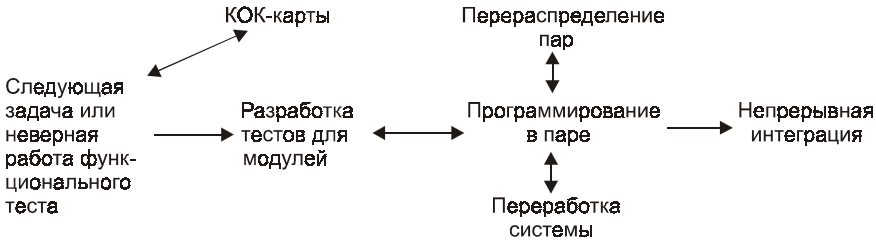


Рис. 3.21. Коллективное владение кодом

Добавим еще, что экстремальное программирование уделяет значительное внимание организации офисного пространства, отмечая существенное влияние окружающих условий на работу программистов.

3.5.7.2. Адаптивная разработка

В основу подхода *адаптивной разработки* (Adaptive Software Development — ASD) положены три нелинейные перекрывающие друг друга фазы: обдумывание, сотрудничество и обучение. Автор данного подхода Джим Хайсмит (Jim Highsmith) обращает особое внимание на использование идей из области сложных адаптивных систем.

Результаты планирования (которое само здесь является парадоксальным) в адаптивной разработке всегда будут непредсказуемыми. При обычном планировании отклонение от плана является ошибкой, которую исправляют. При данном подходе отклонения ведут к правильным решениям.

Программисты должны активно сотрудничать между собой для преодоления неопределенности в подходе адаптивной разработки. Руководители проектов должны обеспечить хорошие коммуникации между программистами. Благодаря этому программисты сами находят объяснения на возникающие вопросы, а не ждут их от руководителей.

Обучение является постоянной и важной характеристикой подхода. И программисты, и заказчики в процессе работы должны пересматривать собственные обязательства и планы. Итоги каждого цикла разработки используются при подготовке следующего.

3.5.7.3. Семейство технологических подходов Crystal

Автор *семейства технологических подходов Crystal* — Алистэр Коуберн (Alistair Cockburn) (**http://www.maxkir.com/sd/justintimemethodologyconstruction_RUS.html**). Каждый из подходов строится на двух базовых принципах: "усиленная коммуникация" и "облегченные рабочие продукты". Количество технологических элементов можно уменьшать для того, чтобы чаще выпускать работающие

варианты системы и как можно шире использовать каналы межличностного общения. Оба эти принципа можно легко применять и видоизменять в любом проекте.

В Crystal принято определять (настраивать) место проекта в трехмерном пространстве по следующим измерениям (рис. 3.22).

- *Критичность проекта.* Это степень серьезности последствий дефекта, появившегося в системе. Обычно выделяют четыре градации критичности: угроза человеческой жизни, существенная потеря денег, незначительная потеря денег, неудобство в работе.
- *Количество занятых в проекте сотрудников.* Для градации здесь используется коэффициент роста от 2 до 2,5. При удвоении числа разработчиков тип коммуникации изменяется настолько, что это требует дополнительного согласования и разработки дополнительных механизмов общения. Упомянутые коэффициенты эвристические и базируются на некоторых соображениях, например, сколько людей можно расположить так, чтобы они могли легко услышать друг друга или подойти друг к другу и т. п.
- *Цели, которые требуется достигнуть при организации процесса.* Эти цели являются критериями эффективности. Такими целями могут быть:
  - соответствие продукта часто изменяющимся требованиям;
  - повышение производительности и времени выпуска продукта при приемлемом качестве;
  - обеспечение повышенных гарантий качества продукта.

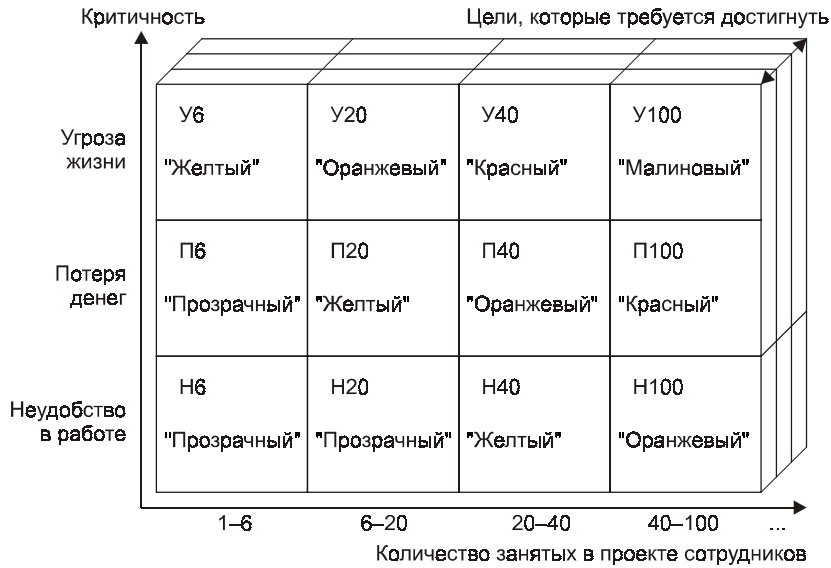


Рис. 3.22. Семейство технологических подходов Crystal

Все подходы Crystal различаются по цвету, причем каждый цвет предназначен для определенных проектов и имеет свои собственные характеристики. "Прозрачный" применяется для самых легких и небольших проектов. Затем идут "Желтый", "Оранжевый", "Красный", "Малиновый", "Синий", "Фиолетовый" для больших команд, использующих все более жесткие технологические подходы соответственно. Методология каждого цвета обладает своими собственными правилами и базовыми элементами. Приведем очень краткие характеристики лишь двух подходов.

Подход Crystal П40 "Оранжевый" предназначен для проектов среднего размера в индустриальном окружении, обладающих следующими характеристиками. Общее количество занятых в проекте людей от 10 до 40, продолжительность работ от 1 до 2 лет, важно своевременное появление продукта на рынке, нужно поддерживать общение между нынешними и будущими разработчиками, а также снижать временные и финансовые расходы. Критичность не жизненно важная. Таким образом, это обычный тип проектов, для которых характерен поиск компромисса между количеством поставляемых компонентов (документации, руководств и пр.) и срочными изменениями в требованиях и дизайне. Подход годится для команды общей численностью не выше 40 человек, работающей в одном здании над разработкой системы, сбой которой может привести к потере несущественной суммы (сюда можно отнести, например, программу учета платежей компании). Характерные особенности подхода следующие.

- Роли включают в себя спонсора, эксперта в данном виде бизнеса, эксперта по использованию системы, технического посредника, аналитика-проектировщика, руководителя проекта, архитектора, проектировщика-программиста, ведущего проектировщика-программиста, инженера по повторному использованию, писателя, тестировщика, дизайнера интерфейсов.
- Все сотрудники распределяются по командам, которые отвечают, соответственно, за планирование, руководство, архитектуру, технологии, функционирование, инфраструктуру и внешнее тестирование.
- Программный продукт проходит процесс контроля качества каждые 3 месяца; прогресс в работе отслеживается по контрольным точкам поставки частей системы, а не по документации; применяется автоматическое регрессивное тестирование функциональности системы. В работах непосредственным образом участвуют пользователи системы, по два пользователя на версию. Вспомогательные виды деятельности начинаются только тогда, когда сама система оказывается достаточно стабильной для ее проверки пользователями.
- "Настройка" подхода осуществляется на собраниях в начале и середине каждого инкремента.
- К рабочим продуктам относятся: документ с требованиями к системе, план выпуска версий, отчеты о состоянии работ по проекту, документ,

описывающий дизайн интерфейсов, общая объектная модель системы, внутрикомандные спецификации, руководство для пользователя, исходный код, тесты и код для миграции с предыдущей системы. Каждый из упомянутых здесь продуктов дорабатывается до той степени, при которой его начнут понимать коллеги по работе, а именно — до уровня детализации, допускающего парную работу.

- Все шаблоны для рабочих продуктов, стандарты кодирования, стандарты построения пользовательских интерфейсов и детали, касающиеся регрессивного тестирования, считаются локальными стандартами, т. е. их должна вырабатывать и поддерживать сама команда разработчиков.

Подход Crystal П6 "Прозрачный" годится для команд общей численностью до 6 человек, сидящих в одной комнате или в одном общем офисе. Когда над проектом работает всего 6 человек, их не нужно подразделять на какие-либо группы, как в случае применения "Оранжевого" подхода. Если люди сидят в непосредственной близости друг от друга, потребуется гораздо меньше усилий, чтобы держать их в курсе дела относительно их участка работ и состояния всего проекта в целом. Характерные особенности подхода таковы.

- Отдельные роли выполняют спонсор, старший дизайнер, дизайнер-программист и пользователь. Все остальные роли распределяются между членами команды: координатор проекта, эксперт в данном виде бизнеса, сборщик требований.
- Над проектом работает только одна команда.
- Стандарты те же, что и в "Оранжевом" подходе, за исключением того, что каждый инкремент длится не более двух месяцев.
- Требуется меньшее количество рабочих продуктов: план выпуска версий, план просмотра системы пользователями и план поставок, снабженные пояснениями варианты использования системы, наброски по дизайну системы и интерфейсам, различные заметки, общая объектная модель системы, работающий код, код миграции, тесты и руководство для пользователя.
- Шаблоны для рабочих продуктов, стиль кодирования, стандарты построения пользовательских интерфейсов и детали, касающиеся регрессивного тестирования, точно так же считаются локальными стандартами, которые должна вырабатывать и поддерживать сама команда разработчиков.

3.5.8. Подходы исследовательского программирования

Исследовательское программирование имеет следующие особенности (**http://www.osp.ru/pcworld/2001/01/062.htm**).

- Разработчик ясно представляет направление поиска, но не знает заранее, как далеко он сможет продвинуться к цели.

- Нет возможности предвидеть объем ресурсов для достижения того или иного результата.
- Разработка не поддается детальному планированию, она ведется методом проб и ошибок.
- Работа связана с конкретными исполнителями и отражает их личностные качества.

В основе исследовательского программирования в большей степени, чем в других подходах, лежит искусство. Это легко объясняется тем, что перечисленные особенности описывают реальную обстановку решения сложных задач, вносящих элементы искусства.

3.5.8.1. Компьютерный дарвинизм

Название данного подхода было предложено Кеном Томпсоном (Ken Thompson). Подход основан на принципе восходящей разработки, когда система строится вокруг ключевых компонентов и программ, которые создаются на ранних стадиях проекта, а затем постоянно модифицируются. Все более крупные блоки собираются из ранее созданных мелких блоков.

Компьютерный дарвинизм представляет собой метод проб и ошибок, основанный на интенсивном тестировании. Причем на любом этапе система должна работать, даже если это минимальная версия того, к чему стремятся разработчики. Естественный отбор оставит только самое жизнеспособное.

Подход состоит из трех основных процессов:

- макетирование (прототипирование);
- тестирование;
- отладка.

Одной из интересных особенностей подхода является максимально возможное распараллеливание процессов тестирования и отладки.

3.5.8.2. Фрагментарное программирование

Особенность этого подхода в том, что сначала создается шаблон программы с работающими кусочками (фрагментами). Далее выполняется постепенное, возможно длительное, приближение к конечной цели. Подход применяется в том случае, когда не могут быть точно сформулированы требования к большей части задачи. У разработчиков есть лишь общее представление о цели, а также идеи по воплощению некоторых фрагментов задачи.

Связующий шаблон играет роль программы-"драйвера", организующей взаимодействие уже написанных фрагментов. Кроме того, шаблон может содержать "заглушки", имитирующие еще не написанные фрагменты и отражающие представление разработчиков об интерфейсе с этими фрагментами.

3.6. Технологии коллективной разработки

Все множество разработок в зависимости от количества участников и типов взаимоотношений между ними может быть сведено к триаде разработок, приведенной на рис. 3.23.

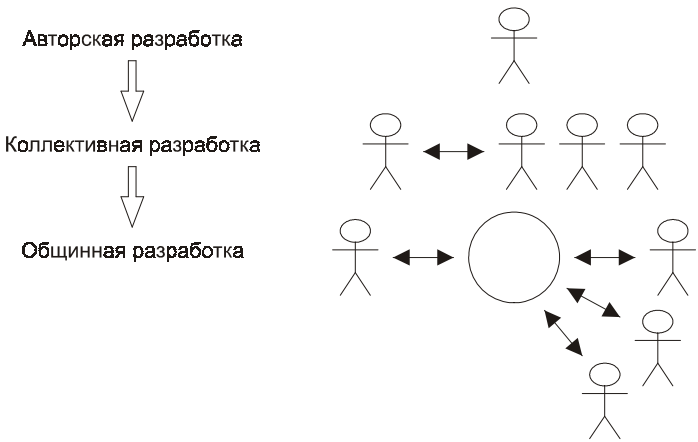


Рис. 3.23. Триада коллективных разработок

3.6.1. Авторская разработка

Люблю одиночество, даже когда я один.
Ж. Ренар

Авторская разработка — принцип создания программных продуктов, при котором весь жизненный цикл разработки поддерживается одним-единственным человеком.

Этот принцип был достаточно широко распространен в 70—80-е годы XX века. Сейчас он применяется редко [Кулаичев 1999]. Примерами авторских разработок являются операционная система Диспак (В. Ф. Тюрин), текстовый редактор Лексикон (Е. М. Веселов), трансляторы с языков Algol-68 (П. Наур) и Pascal (Н. Вирт).

Принцип авторской разработки неприменим для многих современных разработок из-за их сложности, объема и требований к качеству и сопровождению. С другой стороны, программное обеспечение, начиная с момента появления персональных компьютеров, стало продуктом массового потребления, приносящим огромный доход. В этой области быстро выросли и стали доминировать крупные компьютерные компании с развитой структурой менеджмента и мощной рекламой.

Наиболее интересен принцип авторской разработки с точки зрения применения в области наукоемких приложений. Для таких приложений характерна необходимость многолетнего изучения предметной области, практически полное отсутствие начального финансирования проекта, малая рентабельность, определяемая узким кругом пользователей.

> ***Об интеллектуальных работниках***
>
> Отметим, что программисты относятся к интеллектуальным работникам (knowledge workers). Так называют работников, которые могут создавать продаваемый продукт (и, следовательно, зарабатывать себе на жизнь) самостоятельно, вне какой-либо компании. Основным "рабочим оборудованием" таких специалистов (к их числу относятся также юристы и психологи) является их собственная голова.

По данным Алексея Павловича Кулаичева [Кулаичев 1999], авторская разработка может выигрывать по производительности в 30 и более раз у коллективной разработки, что достигается за счет:

- исключения межличностных коммуникаций, связанных с необходимостью порождения и изучения большого количества технологической документации;
- исключения работ по разбиению проекта на составляющие, по распределению их между исполнителями, по координации деятельности исполнителей и контролю за их работой.

Объем программного продукта, выполненного методом авторской разработки, в 5—20 раз меньше по сравнению с индустриальными аналогами.

Авторская разработка предполагает достижение профессионального успеха, известности и славы в одиночку. Такое вполне реально, следует только правильно выбрать профессиональную "нишу", область ведения разработки.

> ***О применении авторской разработки***
>
> В наибольшей степени авторская разработка в наши дни применяется при создании условно-бесплатных программных продуктов (shareware).

3.6.2. Коллективная разработка

Собрать стадо из баранов легко, трудно собрать стадо из кошек.
С. П. Капица

Одним из основных вопросов коллективной разработки является разделение труда — от равноправных соисполнителей до явного и безоговорочного лидера (например, в случае бригады главного программиста).

3.6.2.1. Технические командные роли

Иерархическая модель

Известно, что первые коллективные разработки программ велись примерно так. Начальник выполнял разделение большого проекта на меньшие части и

передавал далее *по иерархии*. Через некоторое время, теперь уже снизу вверх, шла сборка программы из написанных фрагментов. Заметим, что в итоге собрать работающий программный продукт удавалось не всегда.

Матричная модель (равноправные соисполнители)

Бригада *равноправных соисполнителей* обычно состоит из специалистов, занимающихся примерно одинаковыми задачами в рамках одного проекта. Естественно, специализаций в рамках одной бригады может быть несколько. *В разд. 1.7.6.2* уже приводился примерный состав такой бригады разработчиков, а именно:

- инженеры-разработчики (специалисты по инженерии программирования и программисты);
- технические писатели;
- инженеры тестирования;
- инженеры качества;
- специалисты по сопровождению продукта;
- специалисты по продажам продукта.

Тип работы определяет содержание и природу выполняемой работы. Приведем список типов работ и областей специализации на основе классификации Сью Конгер (Sue Conger) [Conger 1994].

- Разработка приложений (Application Development):
  - программист (Programmer);
  - специалист по инженерии программирования (Software Design Engineer);
  - специалист по инженерии знаний (Knowledge Engineer).
- Разработка документации (Documentation):
  - технический писатель (Technical Writer).
- Тестирование продукта (Testing):
  - инженер тестирования (Tester);
  - инженер по разработке тестов (Test Design Engineer).
- Продажи (Sales):
  - консультант по продукту (Product Support);
  - специалист по маркетингу (Product Marketing).
- Управление разработкой (Management):
  - менеджер проекта (Project Manager);
  - менеджер информационных систем (IT Manager).

- Разработка аппаратуры (Hardware Design):
  - разработчик аппаратуры (Hardware Design Engineer).
- Работа с приложениями (Application Support):
  - специалист по приложениям (Application Specialist);
  - администратор данных (Data Administrator);
  - администратор базы данных (Database Administrator).
- Техническая поддержка (Technical Specialists):
  - системный администратор (System Administrator);
  - сетевой администратор (Network Administrator);
  - администратор коммуникаций (Communications Administrator).
- Системное интегрирование (System Integration):
  - системный интегратор (System Integrator).

Из перечисленных специализаций очень интересна специализация *системного интегратора*. Основные задачи системного интегратора — это предложить заказчику вариант решения его проблемы, выбрав наиболее приемлемый по цене и технике, и реализовать его. Таким образом, системный интегратор продает решения и несет ответственность за их реализацию. Системный интегратор как профессионал должен обладать знаниями из очень многих областей: прикладное и системное программное обеспечение, администрирование систем, аппаратура, сети, экономика и т. п.

Бригада главного программиста

— Почему бригада скорой помощи состоит из двух врачей?
— Один знает — куда ставить клизму, а другой — зачем!

Анекдот о специализации в команде

Миллз [Брукс 1999] предложил организовывать *команды (бригады) главного программиста* (chief programmer teams), подобные хирургическим бригадам. Лишь один участник команды занимается основной работой, остальные оказывают ему всевозможную поддержку. Бригада главного программиста включает десять человек, выполняющих специализированные роли в команде (рис. 3.24).

Основные члены бригады выполняют следующие функции.

- *Главный программист.* Лично выполняет анализ и проектирование, создание и отладку кода, написание документации. Должен обладать талантом, большим опытом работы и существенными знаниями.
- *Дублер.* Может выполнять любую работу главного программиста, но менее опытен. Подстраховывает главного программиста, может заниматься написанием кода, но не несет ответственности за проект.
- *Администратор*, он же *менеджер.* Под его контролем находятся деньги, люди, помещения, машинные ресурсы, контакты с другими группами и руководством.

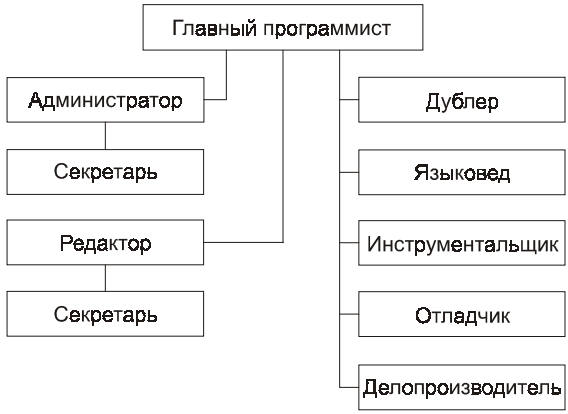


Рис. 3.24. Бригада главного программиста

- *Редактор*. Фактически, это *технический писатель*. Его задача — критически переработать черновики документации, созданные главным программистом, снабдить их ссылками и библиографией и обеспечить публикацию или помещение в Интернете.
- *Языковед*. Эксперт в тонкостях языков программирования. Может найти эффективные способы использования языка для решения сложных задач. Обычно работает с несколькими бригадами.
- *Инструментальщик*. Разработчик специализированных инструментов — утилит и сценариев. Поддерживает основной инструментарий и оказывает по нему консультации. При необходимости может осуществлять администрирование операционной системы.
- *Отладчик*. Разработчик тестов и организатор тестирования программного продукта.
- *Делопроизводитель*. Отвечает за регистрацию всех технических данных бригады в библиотеке программного продукта. Благодаря делопроизводителю, активные программисты освобождаются от рутинных работ. Заметим, что в настоящее время функции делопроизводителя автоматизированы и переданы репозиторию проекта.

Рекомендуется создавать команды на основе бригады главного программиста лишь для небольших проектов. Отметим, что большие трудности могут возникать при поиске кандидатур на роль главного программиста, который должен быть весьма талантливым специалистом. Причем как программистом, так и техническим руководителем.

Программирование в парах

Программирование в парах предусматривает разработку, при которой два человека в одно и то же время занимаются программированием одной задачи

за одним компьютером, используя одну клавиатуру, одну мышь и один монитор. В каждой паре существуют две роли [Бек 2000].

- Первый партнер решает задачу непосредственной реализации одного из методов наилучшим образом. Именно в его руках находится клавиатура и мышь.
- Второй партнер решает стратегические задачи:
  - будет ли работать используемый подход в целом;
  - какими могут быть дополнительные тестовые случаи;
  - существуют ли способы упростить всю систему так, что текущая проблема просто исчезнет.

Состав пар обычно меняется динамически, возможно несколько раз в день.

Ядерная модель

Ядерная модель предполагает наличие первого исполнителя, олицетворяющего ядро команды и создающего прототип системы. На основе прототипа командой разработчиков создается программный продукт. Наиболее сложным действием здесь является передача работы от исполнителя прототипа к команде, которая будет доводить работу до состояния программного продукта.

3.6.2.2. Психологические командные роли

Инструкция для биологов по проведению опытов над мышами:
1. Возьмите мышь.
2. Подготовьте мышь к опыту.
3. Полученную кашицу...

Биологический фольклор

Кроме технических командных ролей следует разбираться и в психологических ролях. С одной стороны, важно понимать характеристики конкретных людей. Это может очень пригодиться, когда надо иметь дело с определенным человеком, и именно для него придумывать способы мотивации. Характеристики должны определяться профессиональным психологом, как правило, на основе интерпретации тестов. Одним из традиционных тестов является Миннесотский стандартный многофакторный метод исследования личности — СМИЛ (Minnesota Multiphasic Personality Inventory — MMPI). По результатам тестирования можно прогнозировать и диагностировать те или иные возможные проблемы или хотя бы области проблем. С другой стороны, важны характеристики людей в отношениях с внешним миром. Такая типология может быть полезной, когда надо решить, кто с какой работой лучше справится. Далее мы рассмотрим два примера командных характеристик.

Ключевые проектные роли

Роб Томсет (Rob Thomsett) [Thomsett 1990] предложил восемь ключевых ролей в проекте (рис. 3.25).

- *Председатель*. Выбирает путь, по которому команда движется вперед к общим целям. Умеет обнаружить сильные и слабые стороны команды и обеспечить наибольшее применение потенциала каждого ее участника.
- *Архитектор*. Он же оформитель. Придает законченную форму действиям команды. Имеет четкое представление о проблемах и их возможных решениях.
- *Генератор идей*. Предлагает радикально новые идеи и стратегии, новые подходы к решению проблем, с которыми сталкивается группа. Особое внимание уделяет главным проблемам.
- *Критик*. Он же скептик, оценивающий проблемы с прагматической точки зрения. Ищет недостатки, изъяны и недоделки. Компенсирует оптимизм генератора идей.
- *Исполнитель*. Работник, собственно занимающийся написанием кода. Как правило, он не обладает широтой кругозора.
- *Завершающий*. Поддерживает в команде настойчивость в достижении цели. Играет доминирующую роль на завершающих стадиях разработки.
- *Дипломат*. Поддерживает силу духа в участниках проекта. Оказывает им помощь в трудных положениях. Пытается улучшить взаимоотношения в команде.
- *Организатор*. Обнаруживает и сообщает о новых идеях, разработках и ресурсах. Имеет много друзей и связей в своей организации, с помощью которых можно выпросить или одолжить необходимые ресурсы.

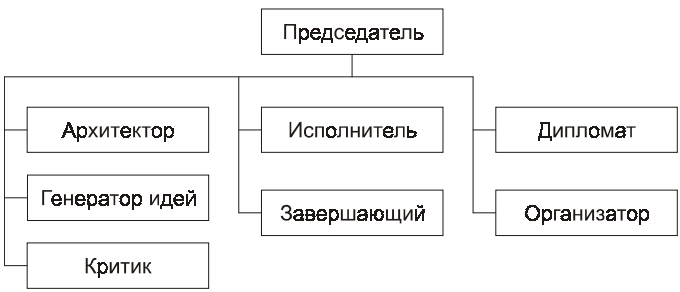


Рис. 3.25. Психологические командные роли

В реальных командах программистов могут быть выделены не все из этих ролей. Роль исполнителя часто берут на себя сразу несколько членов команды.

Соционические роли

Существует много методик для определения психологического портрета человека, а также для помощи при формировании "почти идеальных" малых трудовых коллективов. Для создания и расширения программистских коллективов с максимально продуктивной деятельностью можно применять, например, интересные соционические разработки. При определении прогноза эффективности одной личности и степени совместимости внутри коллектива соционика предлагает использовать легкие и простые при обработке тесты.

Методика основана на выделении концентрированно выраженных и легко узнаваемых типов личности. Основы методики созданы швейцарским психологом Карлом Густавом Юнгом (Carl G. Jung), а развита и усовершенствована она была в работах литовской исследовательницы Аушры Аугустинавичуте [Филатова 1993].

Экстраверт, по определению Юнга, — это человек, чья деятельность направлена на объект и определяется этим объектом. Такой человек имеет тенденцию к направленному взаимодействию с внешней средой. *Интроверт* ориентируется на свою оценку предмета или события, а не на объект как таковой. Юнг выделил четыре базовых области восприятия: материя, энергия, пространство и время (иначе — логика, этика, сенсорика, интуиция). Им соответствуют следующие типы личности: мыслительный, эмоциональный, ощущающий и интуитивный.

В соционике принято выделять восемь основных функций, по которым сознание человека связано с окружающим его миром.

- *Логика*: деловая (экстраверт), отношений (интроверт).
- *Этика*: эмоций (экстраверт), отношений (интроверт).
- *Сенсорика*: волевая (экстраверт), ощущений (интроверт).
- *Интуиция*: возможностей (экстраверт), времени (интроверт).

В каждом типе выделяется два подтипа: мыслительный и эмоциональный, которые имеют сенсорную или интуитивную ориентацию, и наоборот.

На основе знания типа личности могут быть сделаны рекомендации о выборе сферы деятельности (профессиональная ориентация). Выделим четыре типа личности, наиболее склонных к работе, связанной с программированием.

- Интуитивно-логический экстраверт — "впередсмотрящий", импульсивный стратег, склонный к синтезу (характерный представитель типа — Александр Суворов).
- Логико-интуитивный интроверт — сильный логист, имеющий аналитический характер мышления, стремление к выявлению системообразующих факторов (характерный представитель типа — Робеспьер).

- Интуитивно-логический интроверт — критик, стратег, генератор новых алгоритмов, знаток динамических структур (характерный представитель типа — Оноре де Бальзак).
- Логико-интуитивный экстраверт — активный инициатор, изобретатель и рационализатор (характерный представитель типа — Джек Лондон).

3.6.2.3. Типы совместной деятельности

Коллективная разработка предполагает большое количество различных действий, причем степень совместной деятельности может существенно изменяться от одного действия к другому. Можно выделить четыре *типа совместной деятельности* [Robillard, Robillard 2000].

- *Мандатная* деятельность, обычно представленная формальными собраниями, проводимыми на регулярной основе. Обычно собрания планируются заранее, а присутствие на них обязательно. Статистика показывает, что программисты проводят около 4% своего рабочего времени на собраниях.
- *Созываемая* деятельность, которая имеет место в случае намерения двух или более программистов собраться вместе для решения некоторого технического вопроса. Такие собрания обычно не планируются заранее, и в них участвуют только действительно заинтересованные в решении проблемы программисты. На эту деятельность уходит около 14% рабочего времени.
- *Естественная совместная* деятельность, когда как минимум двое программистов работают над одной и той же задачей одновременно и обмениваются информацией о выполняемой работе. Эта деятельность занимает около 41% рабочего времени.
- *Индивидуальная* деятельность, когда программист работает над задачей, которая не выполняется в то же самое время никаким другим программистом, и поэтому маловероятно его взаимодействие по этому предмету с любыми другими программистами группы. Эта деятельность занимает также около 41% рабочего времени.

3.6.3. Общинная модель разработки

> Совершенство в проекте достигается не тогда, когда нечего добавить, а тогда, когда нечего убрать.
>
> *Антуан де Сент-Экзюпери*

Идеология *общинной ("базарной") модели* разработки сформулирована в программной статье Эрика Раймонда (Eric Raymond) "Собор и Базар" (**http://www.osp.ru/os/1999/09-10/071.htm**). Общинная модель характеризуется тремя основными факторами.

- *Децентрализованность разработки.* Не существует ограничения сверху на количество людей, принимающих участие в проекте. Как правило,

разработки такого типа ведутся на базе сети Интернет и могут включать любого заинтересованного разработчика.

- Разработка *на базе открытых исходных текстов*. По ним можно разобраться с сутью задачи и в любой момент подключиться к разработке.
- *Большое количество внешних тестеров* (бета-тестеров), позволяющих быстро обнаруживать ошибки и проблемы в программе.

Эрик Раймонд сформулировал несколько уроков, которые позволяют лучше понять особенности общинной разработки.

- Каждая хорошая программа начинается с энтузиазма разработчика.
- Хорошие программисты знают, что можно написать, а великие — что можно переписать.
- При правильном отношении интересная проблема найдет вас сама.
- Когда вы теряете интерес к программе, ваша последняя обязанность — передать ее компетентному преемнику.
- Следует выпускать ранние и частые версии программ.
- Обнаружить проблему и исправить ее могут разные люди.
- Иногда использовать идеи пользователей лучше, чем свои.

В сети Интернет можно найти достаточно большое количество сайтов с проектами, разрабатываемыми по общинной модели, например **http://www.gnu.org** и **http://sourceforge.net**.

3.6.4. Отступление "об оффшорном программировании"

Оффшорное программирование — это выполнение внутренних работ компании сторонними специалистами, вне ее офиса и, как правило, на территории другой страны. Эта разновидность коллективного программирования получила известность в начале 1990-х годов. Формы предоставления услуг оффшорного программирования прошли следующие этапы эволюции.

- *Аутстаффинг* — использование программистов "поштучно" для конкретной работы под руководством менеджера заказчика.
- *Аутсорсинг* — передача исполнителю разработки отдельных модулей и компонентов систем, полная сборка которых производится заказчиком.
- *Полная разработка* — выполнение проекта по полной разработке и внедрению системы.

Довольно широкое распространение оффшорного программирования в настоящее время обусловлено состоянием мирового рынка заказного программного обеспечения. По очень приблизительным данным разработкой программного обеспечения в мире занято от 7 до 20 миллионов человек.

В России — от 5 до 10 тысяч. В мире существует огромный неудовлетворенный спрос на услуги профессионального программирования (например, в США дефицит профессиональных программистов составил в 2003 году около 1,5 миллионов). Следовательно, для некоторых компаний передача части работ исполнителям в другой стране обусловлена естественной необходимостью.

Поскольку стран, в которые могут быть переданы оффшорные заказы, достаточно много, мы можем сформулировать для них три основных желательных условия.

- Оплата профессионалов ниже, чем в стране-заказчике.
- Присутствуют высокие стандарты образования и доступны технические эксперты.
- Доступны передовые технологии, повышение технической квалификации.

Как легко видеть, основные страны, удовлетворяющие данным условиям, — это Россия, Индия, Китай. Среди лидеров оффшорного программирования также находятся Уругвай и Израиль.

Некоторый идеальный портрет оффшорного программиста и оффшорной компании может выглядеть следующим образом.

- Высокое качество работы каждого сотрудника. В оффшорных компаниях в первую очередь "люди решают все". На уровне компании должны быть созданы системы отбора и оценки сотрудников, системы карьерного роста, обучения, мотивации, социальной защиты и решения многих других задач.
- Высокий уровень ведения проекта. В основе любого проекта лежат несколько ключевых процессов, грамотное выполнение которых должен обеспечить менеджер проекта.
- Оптимальная структура управления компанией. Такая структура подразумевает возможность быстрого формирования и переформирования команды, а также отсутствие жесткой и сложной иерархии.

В завершении краткого обзора оффшорного программирования укажем положительные и отрицательные стороны создания оффшорных компаний в России. Конечно, положительных сторон много, и они достаточно хорошо известны. В первую очередь, это фундаментальная подготовка, позволяющая браться за большие проекты *(см. разд. 1.3 и 1.4)*. Немаловажную роль может играть территориальная близость и близость культурных сред. Слабые стороны мы явно подчеркнем в следующих тезисах.

- Небольшое количество профессионально подготовленных менеджеров, способных грамотно управлять программными проектами. Большинство компаний, передающих заказы, считает это основной проблемой, препятствующей резкому росту оффшорного программирования в России.

- Недостаточное знание естественного языка той страны, из которой поступает работа. Последнее время ситуация начинает меняться к лучшему благодаря росту интереса к иностранным языкам в школах и их грамотному преподаванию.
- Большие сложности поездок в страну фирмы, передавшей заказ. Известно много случаев отказа в выдаче виз по надуманным причинам инженерам, выезжающим на короткий срок для осуществления консультаций и участия в совещаниях рабочих групп. Особенно "славятся" такой политикой США.
- Относительно высокая стоимость качественных телекоммуникационных услуг. Это может стать причиной дополнительных расходов.
- Низкий уровень сертификации на соответствие стандартам качества. Для ряда зарубежных компаний наличие такого сертификата у российской компании, в которую передается заказ, является обязательным.
- Повышенная осторожность зарубежных клиентов при взаимодействии с российскими компаниями. В ряде случаев эта осторожность обусловлена историческими и политическими причинами. В других случаях — предыдущим неудачным опытом, связанным, например, с традиционным российским упованием на "авось".

> ***О проблеме русских диаспор***
>
> Замечено, что передача заказов в одну из стран в большой степени определяется наличием в компании, передающей работу, граждан этой страны, работающих по контракту. В наибольшей степени лоббированием передачи заказов в свою страну занимаются индусы. Как это ни печально, российские специалисты обычно не спешат с поддержкой действий по передаче работ в Россию.

3.7. Качество программного обеспечения

Мы должны отыскать себе благородного мужа, которого мы имели бы всегда перед глазами, чтобы жить, точно он всегда смотрит на нас, и поступать, точно он видит наши поступки.

Эпикур

Завершим данную главу рассмотрением вопроса качества программного обеспечения. Неявно подразумевается требование идеального качества. Одна из возможных альтернатив качественному программному обеспечению — "достаточно хорошее" программное обеспечение, которое мы также рассмотрим. Программисты-профессионалы, прекрасно понимая необходимость создания качественного программного обеспечения, видят некоторую однобокость исследований. Да, сертифицированная по ISO 9000 организация не должна потерять исходный код программ, работающих у заказчика. Однако стандарт не дает ничего позитивного в улучшении способности программировать.

3.7.1. Подходы к качеству программного обеспечения

Проведем классификацию различных подходов к качеству программного обеспечения, используя два измерения [Vliet 2000].

- Первое измерение ориентировано либо на продукт, либо на процесс. Для повышения качества программного обеспечения можно сосредоточиться на качестве самого продукта, например, делая его более дружественным пользователю. Альтернативный подход заключается в совершенствовании процесса разработки, предполагая при этом, что чем лучше процесс, тем лучше качество программного обеспечения.
- Второе измерение связано либо с соответствием, либо с усовершенствованием. Под соответствием будем понимать соответствие какому-либо стандарту. Усовершенствование имеет своей целью переход на более совершенные методы и лучшую практику для повышения качества.

В табл. 3.2 приведены примеры каждого из четырех подходов:

- ISO 9126 является стандартом на качество продукта, определяющим атрибуты и характеристики качества, включая измерения количественной оценки этих характеристик;
- "усовершенствованием практики", например, является усовершенствование управления конфигурацией программного обеспечения, инспекций, тестирования и т. п.;
- ISO 9000 [ISO 9000 1992] — это совокупность стандартов, декларирующих требования для качественных систем. С точки зрения разработки программного обеспечения наиболее полезны "Руководящие указания по применению ISO 9001 при разработке, поставке и обслуживании программного обеспечения" [ISO 9001 1992];
- методы усовершенствования процесса разработки программного обеспечения предлагают некоторую шкалу уровней и требования соответствия, согласно которым можно определить место компьютерной компании на этой шкале. Наиболее известны и популярны два метода:
  - модель зрелости процессов разработки программного обеспечения (Capability Maturity Model for Software — CMM), предложенная Software Engineering Institute (SEI);
  - определение возможностей и улучшение процессов создания программного обеспечения (ISO/IEC 15504 Software Process Improvement and Capability determination — SPICE).

Таблица 3.2. *Подходы к качеству*

Объект качества	Соответствие	Усовершенствование
Продукт	ISO 9126	"Усовершенствование практики"
Процесс	ISO 9000, процесс обеспечения качества	CMM, SPICE, ...

Два важнейших утверждения лежат в основе достижения качества.

- Качество начинается с удовлетворения потребностей разработчиков.
- Качество доказывается удовлетворением потребностей пользователей.

Подходы к достижению качества таковы:

- качество достигается с помощью квалифицированных разработчиков, точного соблюдения процессов и удачных технологических подходов;
- качество достигается путем полного понимания всех действий и изменений. Ни одна строка в программе не должна быть ни добавлена, ни изменена без полного понимания — что, зачем и как делается;
- качество достигается путем тщательного тестирования программы перед тем, как она будет доступна пользователю;
- достижение качества должно планироваться;
- достижение качества — обязанность каждого разработчика.

Конечно, качество должно быть не только достигнуто, но и сохранено в процессе развития продукта.

3.7.2. Характеристики качества программного обеспечения

> В настоящее время не существует общепринятых критериев качества программного обеспечения.
>
> *Стандарт ISO 9000-3, п. 6.4.1*

Классическое определение *качества* заключается в том, что разработанный программный продукт подтверждает свою спецификацию, при этом спецификация должна быть ориентирована на характеристики, которые желает получить клиент.

Современные стандарты уточняют понятие качества, вводя совокупность черт и характеристик, которые влияют на его способность удовлетворять заданным потребностям пользователей. Перечислим ряд таких характеристик [Жоголев 1996].

- *Функциональность* (пригодность, точность, интероперабельность, согласованность, безопасность). Функциональность — это способность

программного продукта выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор таких функций определяется во внешнем описании программного продукта.

- *Надежность* (завершенность, устойчивость, восстанавливаемость).
- *Удобство* (понимаемость, эффективность освоения, эргономичность). Удобство — это характеристика программного продукта, которая позволяет минимизировать усилия пользователя по подготовке исходных данных, применению программного продукта и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя.
- *Эффективность* (по времени и по ресурсам). Эффективность — это отношение уровня услуг, предоставляемых программным продуктом пользователю при заданных условиях, к объему используемых ресурсов.
- *Сопровождаемость* (простота анализа, изменяемость, стабильность, проверяемость). Сопровождаемость — это характеристика программного продукта, позволяющая минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.
- *Переносимость* (адаптируемость, гибкость инсталляции, согласованность со стандартами и правилами, заменяемость). Переносимость — это способность программного продукта быть перенесенным из одной среды в другую, в частности, с одной аппаратной архитектуры на другую.
- *Добротность* (рациональная организация, продуманность, непереусложненность).

Две наиболее интересные характеристики рассмотрим подробнее.

Надежность — это способность программы безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. Надежный программный продукт не исключает наличия в нем ошибок. Здесь важно, чтобы ошибки при практическом применении в заданных условиях проявлялись достаточно редко. Степень надежности характеризуется вероятностью работы программного продукта без отказа в течение определенного периода времени.

Существуют следующие подходы по обеспечению надежности:

- предупреждение ошибок;
- самообнаружение ошибок;
- самоисправление ошибок;
- обеспечение устойчивости к ошибкам.

Добротность программы заключается в том, что программа разумно и рационально организована, с достаточно продуманной организацией потоков

управления и информационных потоков, не слишком переусложнена. Понятие добротности введено Поттосиным [Поттосин 1997] для оценки внутренних достоинств реализации программы с технической стороны. Поттосин вводит четыре класса критериев добротности программ.

- *Количественные* критерии, связанные с различными способами оценки (метриками) сложности программ. Приведем примеры численных характеристик.
  - Меры Холстеда [Холстед 1981], включающие ряд формул, оценивающих длину, объем, уровень и интеллектуальное содержание программ.
  - Оценка сложности управляющего графа программы. Фрагмент программы может быть оценен цикломатическим числом ее управляющего графа, которое равно `m – n + 2`, где `m` — число дуг, а `n` — число вершин управляющего графа. Считается, что цикломатическое число не должно превышать 10.
  - Оценка модульного разбиения программы. Такая оценка должна состоять из множества критериев. Например, сложность модуля оценивается совокупностью сложности определяемых в нем процедур и сложности связей модуля с другими модулями по импорту и экспорту определяемых сущностей.
- *Генетические* критерии, связанные с происхождением программы и дисциплиной ее создания.
- *Структурные* критерии, связанные с оценкой организации управления в программе и отражением организации управления в программном тексте.
- *Прагматические* критерии, связанные с оценкой того, насколько программный текст соответствует цели программы. Формулируется список излишеств, которых не должно быть в добротных программах, например вычислительной избыточности.

3.7.3. Оценка качества процесса разработки

> Требовать и эффективности, и гибкости от одной и той же программы —
> все равно, что искать очаровательную и скромную жену.
> По-видимому, нам следует остановиться на чем-то одном из двух.
>
> *Д. Вейнберг*

В начале *разд. 3.7* мы отметили, что существует два подхода, которые могут применяться для оценки качества программного продукта.

- Оценить качество конечного продукта.
- Оценить качество процесса разработки.

Оценить качество конечного продукта можно тестированием и эксплуатацией. На это должно быть отведено время после завершения основной работы над программой. А вот второй подход должен стать частью долговременной стратегии компании.

3.7.3.1. Модель зрелости процессов разработки программного обеспечения (CMM)

Начиная с1991 года, были разработаны *модели зрелости CMM* для различных областей применения, причем наиболее существенными из них были:

- модель зрелости процессов разработки программного обеспечения (Capability Maturity Model for Software — SW-CMM);
- модель зрелости процессов для системного реинжиниринга (Electronic Industries Alliance Interim Standard — EIA/IS 731);
- модель зрелости процессов интегрированной разработки продуктов (Integrated Product Development Capability Maturity Model — IPD-CMM).

На основе этих моделей был построен стандарт CMMI (Capability Maturity Model Integration) (**http://www.sei.cmu.edu/cmm/cmms/cmms.html**). Он вобрал в себя лучшее из моделей, устранив неоднозначность некоторых понятий ввиду наличия множества моделей.

Определим *уровень зрелости* (maturity level) как точно определенное эволюционное плато на пути к достижению полной зрелости процессов организации. Каждый уровень зрелости формирует отдельный слой фундамента для постоянного совершенствования производственного процесса, включает в себя набор целей процесса, которые, по мере их достижения, приводят к стабилизации значимых компонентов производственного процесса. Каждый уровень зрелости является основой для более высокого уровня зрелости. Каждый уровень зрелости состоит из определенных областей процессов. В модели определено пять уровней зрелости организации.

- *Начальный уровень.* На этом уровне процесс разработки характеризуется практическим отсутствием процессов управления. Успех проекта зависит от индивидуальных усилий, личных качеств и даже героизма участников проекта.
- *Повторяемый уровень.* На этом уровне зрелости в компании должны быть внедрены основные процессы управления для отслеживания стоимости, графика проекта и его функциональности. Уровень характеризуется тем, что управление проектами основывается на накопленном опыте и ранее достигнутые успехи будут повторены на подобных приложениях.
- *Определенный уровень.* Процесс разработки программного обеспечения (как на уровне управленческой, так и инженерной деятельности) документирован, стандартизован и интегрирован на уровне всей организации. Процесс перестает зависеть от индивидуальных качеств отдельных разработчиков и не может скатиться на более низкие уровни в кризисных ситуациях.
- *Управляемый уровень.* В компании устанавливаются детальные количественные показатели на процесс разработки и качество продукта. И процесс разработки, и продукты понимаемы и контролируемы.

- *Оптимизирующий уровень*. Продолжается совершенствование процесса разработки на основе анализа текущих результатов процесса и применения инновационных идей и технологий.

Уотс Хамфри (Watts Humphrey), сотрудник SEI, активно участвовавший в разработке модели CMM, разработал *персональный процесс разработки* программного обеспечения (Personal Software Process — PSP) (**http://www.sei.cmu.edu/publications/documents/00.reports/00tr022.html**), который дает возможность программистам значительно повысить качество создаваемого программного обеспечения.

3.7.3.2. Определение возможностей и улучшение процессов создания программного обеспечения (SPICE)

Данная модель очень близка к модели зрелости, но уровни возможностей могут быть применены не только к организации в целом, но и к отдельно взятым процессам. Модели такого типа часто называют *непрерывными*. В непрерывных моделях один процесс может находиться на низком уровне зрелости, а другой — на высоком уровне. Стандарт определяет шесть уровней зрелости процесса (**http://www.sqi.gu.edu.au/spice**).

- Уровень 0. Процесс не выполняется.
- Уровень 1. Выполняемый процесс.
- Уровень 2. Управляемый процесс.
- Уровень 3. Установленный процесс.
- Уровень 4. Предсказуемый процесс.
- Уровень 5. Оптимизирующий процесс.

Во время оценки и улучшения качества процессов выполняются следующие задачи [Терехов, Туньон 1999].

- Сравнение процесса разработки программного обеспечения, существующего в данной организации, с описанной в стандарте моделью. Анализ результатов дает возможность определить сильные и слабые стороны процесса, его внутренние риски.
- Оценка возможности улучшения данного процесса на основе определения текущих возможностей.
- Техническая реализация поставленных задач на основе сформулированных целей совершенствования процесса. После этого весь цикл работ начинается сначала.

О роли Министерства обороны

Заметим, что исторически модели оценки качества процесса разработки создавались с целью получения обоснованных процедур оценки и последующего развития технологических процессов в тех организациях, которые претендуют

на получение заказов по разработке проектов, имеющих оборонное значение. Модели применимы к компаниям, разрабатывающим сложные и работающие в режиме реального времени системы. Это системы с длительным временем жизненного цикла, где дефекты в программном обеспечении могут иметь критическое значение.

3.7.4. "Достаточно хорошее" программное обеспечение

Вчера в Сиэтле после упоминания Биллом Гейтсом о выходе бета-версии новой программы компании Microsoft произошло землетрясение. Пользователи с ужасом ждут объявления о выходе финальной версии продукта.

Анекдот по мотивам землетрясения и выступления Б. Гейтса 28-го февраля 2001 г.

Пропагандистом *"достаточно хорошего" программного обеспечения* является Эдвард Йордон [Йордон 2001]. Подчеркнем, что он применяет это понятие к "безнадежным проектам", которые связаны жесткими ограничениями на время, бюджет и людские ресурсы *(см. разд 1.6)*. В большинстве безнадежных проектов заказчик может быть удовлетворен системой, которая достаточно дешева, достаточно производительна, обладает необходимыми возможностями, достаточно устойчива и будет готова достаточно скоро. Фактически, эти характеристики можно считать определением "достаточно хорошего" программного обеспечения.

Оказывается, что даже "достаточно хорошее" программное обеспечение создать сложно. Приведем часть из совокупности причин, дающих этому объяснение [Йордон 2001].

- Часто качество стремятся определить только в терминах дефектов (ошибок). С точки зрения пользователя не менее важны такие стороны, как готовность к определенной дате.
- Предполагается, что малое количество ошибок равносильно лучшему качеству, и пользователь предпочитает именно такое качество. Однако иногда пользователь готов идти на компромиссы и примириться с некоторыми ошибками в обмен на более скорое выполнение работы.
- Игнорируются такие факторы, как моральное состояние команды разработчиков и адекватные условия для работы.

Джеймс Бах (James Bach) [Bach 1995] указывает следующие необходимые условия для создания "достаточно хорошего" программного обеспечения:

- утилитарная стратегия — искусство количественного анализа и максимизации чистого выигрыша в неопределенных ситуациях;
- эволюционная стратегия, рассматриваемая не только по отношению к жизненному циклу проекта, но также к людям, процессам и ресурсам;

- героические команды — не могучие гениальные программисты, а обычные умелые специалисты, способные к эффективному сотрудничеству;
- динамическая инфраструктура — противоположность бюрократии и засилью политики;
- динамические процессы — процессы, поддерживающие работу в эволюционной среде.

К сожалению, "достаточно хорошее" программное обеспечение редко бывает действительно достаточно хорошим. Никлаус Вирт отмечает, что это "следствие грустного проявления духа нашего времени, в котором исчезает личная гордость за свою работу". По его мнению, "нельзя ожидать качественно выполненной работы, если не отдавать ей себя полностью, если нет личного удовлетворения, более того — удовольствия от нее".

Интересное наблюдение, заключающееся в том, что некоторые компании стремятся к понижению интеллектуального уровня пользователей своей продукции, сделано многими программистами. Компаниям чрезвычайно выгодно иметь дело с людьми, техническая квалификация которых не позволяет определить реальные аспекты (например, качество, сложность, ценность) программного продукта. Прикрываясь "упрощением" работы и повышением доступности компьютеров для пользователей, компании многократно перегружают и усложняют программное обеспечение таким образом, чтобы пользователю было крайне трудно понять, каким образом оно работает на самом деле и стать профессионалом.

3.7.5. Стандартизация информационных технологий

Стандарт — общепринятое определение компонента технических или программных средств, являющихся результатом соглашения. *Профиль* — набор юридических и/или фактических стандартов, ориентированных на выполнение конкретной задачи [Козлов 1999].

Стандарты можно классифицировать следующим образом:

- по типу установления требований:
  - устанавливающие требования к объекту;
  - устанавливающие требования к процессу;
- по масштабу:
  - международные;
  - государственные;
  - отраслевые;
  - предприятий;

- по степени юридического оформления:
  - принятые юридически;
  - действующие фактически.

Процесс стандартизации информационных технологий поддерживают три основные группы организаций **http://www.citforum.ru/programming/prg96/sukhomlin.shtml**).

- Международные организации, входящие в структуру ООН.
  - International Organization for Standardization (ISO) — международная организация по стандартизации.

Об ISO

В 1947 году представители 25 стран решили создать организацию, основной задачей которой стала бы координация разработок и унификация международных стандартов. Новая организация получила название International Organization for Standardization (ISO). Несоответствие полного названия и аббревиатуры объясняется тем, что "ISO" — это греческий префикс, означающий "равный".

  - International Electrotechnical Commision (IEC) — международная электротехническая комиссия.
  - International Telecommunication Union-Telecommunications (ITU-T) — международный союз по телекоммуникации — телекоммуникация. До 1993 года эта организация называлась International Telegraph and Telephone Consultative Committee — международный консультативный комитет по телефонии и телеграфии.
- Промышленные профессиональные или административные организации.
  - Institute of Electrical and Electronic Engineers (IEEE) — Институт инженеров по электротехнике и электронике.
  - Internet Activity Board (IAB) — совет управления деятельностью Интернета.
- Промышленные консорциумы.
  - Object Management Group (OMG) — группа управления объектами.
  - X/Open — консорциум, организованный группой поставщиков компьютерной техники.
  - Open Software Foundation (OSF) — фонд открытого программного обеспечения.

В 1987 году ISO и IEC объединили свою деятельность в области стандартизации информационных технологий и создали единый орган — Joint Technical Committee 1 (JTC1) — объединенный технический комитет 1. Этот комитет предназначен для формирования системы базовых стандартов в области информационных технологий.

Об абсолютной стандартизации

Идеальный подход мог бы состоять в разработке исчерпывающего набора стандартов. Однако для некоторых современных направлений в области информационных технологий это нереально, учитывая скорость (а также стоимость) происходящих изменений. Ко времени внесений очередной порции изменений может появиться новая версия, или программистское сообщество может вообще начать двигаться в другом направлении. Абсолютная стандартизация — это мираж.

3.8. Заключение

3.8.1. Выводы и рекомендации

Дать рецепт для дистанционного лечения невозможно, поскольку каждый больной уникален. По слухам, слово "галиматья" происходит от фамилии французского врача XIX века, лечившего по переписке. Можно дать только общие советы на основе прошлого опыта.

И. Ашманов

Вдумчивый читатель, прочитав главу и добравшись до этого места, может заметить, что ее можно озаглавить "33 источника и 33 составные части технологических подходов". Причем существует и большое число непустых подмножеств 33-элементного множества. Все эти подходы в действительности основаны на небольшом числе основных идей, по-разному комбинируемых. Предложим общие рекомендации по выбору технологического подхода.

Разные проекты должны поддерживаться разными технологическими подходами. В качестве критериев, отличающих проекты, могут быть выбраны требования к создаваемому программному продукту, риски разработки, размер проекта, квалификация программистов и т. п. Выбор должен начинаться между жесткими и гибкими подходами. Кратко напомним их основные особенности.

- *Гибкие подходы* основаны на межличностных коммуникациях. Их главное достоинство в том, что большие объемы информации обрабатываются в результате неформального и непосредственного личного общения. Гибкие подходы с пониманием относятся к тому, что программист делает ошибки, и дают возможность обнаружить их как можно раньше. Гибкие подходы идеальны в том случае, если требования к программному продукту подвержены частым изменениям.
- *Жесткие подходы* предписывают обрабатывать большие объемы информации с помощью документации. Подходы формализуют прогресс проекта, требуя создавать документы и утверждать их перед началом непосредственной работы. Это дает формальные гарантии того, что продукт обладает теми или иными качествами. В случае жестких подходов имен-

но процесс помогает программисту делать меньше ошибок. Жесткий подход необходим в случае большого количества программистов, работающих над проектом, когда непосредственное общение затруднено.

Таким образом, и гибкие, и жесткие подходы ориентированы на то, чтобы создавать высококачественный программный продукт, но делают они это по-разному. В любом случае, еще раз подчеркнем близость гибких и жестких подходов в их целях.

Формальные подходы менее распространены и менее популярны среди разработчиков программных продуктов. Это можно объяснить, с одной стороны, большими ресурсами, требующимися на их поддержку. С другой стороны, многие такие подходы требуют специального инструментария, которого практически не существует. С третьей стороны, с сожалением заметим, что теоретическая база подходов до сих пор далека от совершенства. Очень перспективным является технологический подход сборочного программирования, однако в большинстве случаев отсутствуют (или далеки от принятия) стандарты на различные модули (объекты). Для больших компаний, тем не менее, следует активно создавать типичные модули и повторно использовать их в проектах. Покупка модулей тоже является хорошим возможным решением, но их не всегда удается легко интегрировать в собственную систему.

Если говорить о советах по выбору конкретного технологического подхода, то их два.

- Первый совет состоит в том, что следует расположить основные классы подходов (а можно и конкретные подходы) на шкале в диапазоне от гибких до жестких (рис. 3.26). Далее следует попытаться определить, в какой степени будущий проект потребует тех особенностей гибких и жестких подходов, которые мы указали выше. Результатом должна стать точка на этой шкале, связываемая с некоторым классом подходов.
- Второй совет отсылает нас к достаточно продуманному семейству технологических подходов Crystal. В нем рекомендуется определять место проекта в трехмерном пространстве в зависимости от следующих факторов:
  - критичности проекта;
  - количества занятых в проекте сотрудников;
  - целей, которых требуется достичь при организации процесса.

В рамках одного класса следует принять решение о выборе конкретного подхода. Например, в случае класса адаптивных технологических подходов может возникнуть выбор между экстремальным программированием и семейством технологических подходов Crystal. После проведения краткого сравнительного анализа становится ясно, что первый подход предпочтительнее для небольших коллективов, отличающихся высокой (более того,

жесткой) дисциплиной. Второй подход не требует соблюдения большого количества правил, легко и гибко настраивается на разные проекты.

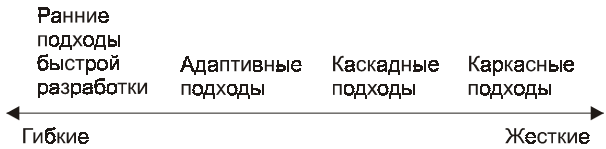


Рис. 3.26. Шкала основных классов технологических подходов

Важный (и не столь очевидный) вывод заключается в том, что в процессе жизненного цикла разработки можно переходить с одного технологического подхода на другой. Технологический подход может изменяться в ходе работы над проектом и приспосабливаться к его нуждам. Это еще называется настройкой на подход, которая обычно производится по достижении очередного этапа. Типичный случай изменения технологического подхода имеет место, когда после получения первого представления о системе заказчик увеличивает финансирование проекта и одновременно требует большей жесткости в сроках и качестве. В этом случае, например, можно перейти с эволюционного прототипирования на экстремальное программирование.

Следует обращать внимание на тесную связь с заказчиком и подключение его к оценке программного продукта на ранних стадиях его разработки. Здесь хороший пример показывают гибкие технологические подходы, стремящиеся реализовать базовую функциональность как можно скорее. На основе первых версий или даже демонстрационного прототипа заказчик получает представление о будущей системе, что позволяет принять верные решения о направлениях ее развития.

В условиях постоянного кризиса программирования *(см. разд. 1.6)* важно не только *на каком языке программирования* разрабатывается программа, не только *как* это делается, а, в первую очередь, *кто* программу создает. В любых проектах — и малых и больших — люди являются ключевым фактором, и никакие формальные технологические подходы ведения проекта не смогут заменить работу с людьми. Во многом отношения между людьми будут определять успех проекта. Обратим внимание на то, что ни один существующий на данный момент технологический подход не учитывает специфику культурной среды разработчиков.

Отметим, что стандартизация по ISO 9000 или CMMI в большинстве случаев не дает какого-либо реального резкого улучшения качества, но дает существенные преимущества в получении заказов, поскольку заказчик ориентируется на наличие у компании сертификатов о соответствии.

Наконец, что же делать, если проект явно не укладывается в отведенные ресурсы, например временные? Закон Брукса утверждает, что выделение

дополнительных людей для отстающего от графика проекта лишь задерживает его выполнение. Тем не менее, практика программирования показывает, что добавление новых людей в ряде случаев позволяет существенно ускорить завершение его выполнения (правда, при этом существенно растут финансовые расходы). Многочисленные исследования в этой области показывают, что данную формулировку следует дополнить тщательно сделанными соответствующими оговорками. Например, о том, что дополнительные программисты, привлекаемые на поздних стадиях, должны не пытаться изменить процесс, а попробовать вписаться в него.

3.8.2. Проблемы и перспективы развития

Человека надо уважать уже хотя бы за его намерения,
потому что все равно у него ничего не получится.
Михаил Задорнов

В настоящее время развитие и совершенствование технологических подходов ведется по двум направлениям, которые достаточно сильно отличаются друг от друга своими задачами.

- *Максимизация качества.* Задачи, решаемые на этом направлении, — надежность, четкость и формализация — ориентированы на военные разработки и системы реального времени. Данное направление поддерживается строгими технологическими подходами, обеспечивающими предсказуемость. Проблеме качества был посвящен *разд. 3.7.*

- *Максимизация скорости разработки.* Это направление в большей степени обращено к искусству, импровизации и поиску. Гибкие и адаптивные технологические подходы поддерживают именно этот путь. Особенность разработки определяется появлением новых классов систем. За последние несколько десятков лет такими классами были, например, системы "клиент-сервер", распределенные системы, интерактивные системы, системы интернет-приложений. С точки зрения технологий, задача разработки интернет-приложений является очень специфичной. Базироваться на классических технологических подходах ей не позволяют следующие особенности:

  - задача, как правило, не будет четко определена и специфицирована;
  - заказчики не смогут точно сформулировать свои требования;
  - как только система будет завершена, она устареет;
  - система должна правильно взаимодействовать с независимо разработанными программами.

Еще одно важное перспективное направление, связанное с технологией программирования, — исследование человеческих и социальных факторов в информатике и программировании.

Предполагается, что в ближайшее время основные усилия в исследовании разработки программного обеспечения будут направлены на процесс проектирования. Это объясняется тем, что в больших проектах основной объем работ приходится на проектирование, которое должны выполнять талантливые и творческие специалисты.

3.8.3. Рекомендации по литературе

Книги банальны. Гениальна только жизнь.
Томас Карлейль

По технологии программирования издано очень большое количество книг. Особого внимания заслуживают пять из них.

- "Мифический человеко-месяц, или как создаются программные системы" [Брукс 1999]. Считается, что без прочтения этой книги не может состояться ни один крупный руководитель программного проекта. Первое издание книги вышло в 1975 году. Спустя 20 лет Фредерик Брукс выполнил ревизию своих старых идей и добавил новые.
- "Инженерия программного обеспечения" [Соммервилл 2002]. Эта книга считается классическим учебником по технологии программирования.
- "Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте" [Йордон 2000]. Книга написана профессионалом, понимающим все современные проблемы и сложности в создании программного продукта.
- "Объектно-ориентированный анализ и проектирование с примерами приложений на C++" [Буч 1998]. Это прекрасное практическое руководство по созданию объектно-ориентированных систем. Считается, что благодаря этой работе объектно-ориентированное проектирование и система обозначений были взяты тысячами программистами для применения в реальных разработках в различных предметных областях.
- "Проектирование программного обеспечения экономических информационных систем" [Вендров 2000]. Особенность этой книги заключается в изложении международных и отечественных стандартов на процессы жизненного цикла программного обеспечения. Достаточно подробно описан ряд промышленных технологий.

Часть работы "Методология программирования" [Турский 1981] посвящена проектированию программ. В книге "Программное обеспечение и его разработка" [Фокс 1985] автор излагает общие проблемы создания программного

обеспечения, его сопровождения и использования с обилием конкретного материала (правда, несколько устаревшего). Рекомендации по управлению разработкой программного продукта подробно изложены в работе "The New Software Engenering" [Conger 1994]. Большое количество метрик для оценки качества проектных решений приведено в учебном пособии "Технологии разработки программного обеспечения" [Орлов 2003].

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Программная инженерия" (Software Engineering). Основные разделы этой темы следующие:
  - проектирование программного обеспечения;
  - процессы разработки, жизненный цикл программного обеспечения и модели процесса разработки;
  - спецификации и требования к программному обеспечению;
  - проверка соответствия программного обеспечения;
  - эволюция ПО Сопровождение программного обеспечения;
  - управление программными проектами;
  - надежность программного обеспечения.
- Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет следующие дисциплины специализации:
  - технология разработки программного обеспечения;
  - метрология и качество программного обеспечения.

3.8.4. Вопросы и задания

1. Обсудите особенности проектирования высококачественного программного обеспечения.
2. Сравните объектно-ориентированный анализ и проектирование со структурным анализом и проектированием.
3. Выберите и примените шаблон проектирования, подходящий для разработки некоторого своего приложения на основе объектно-ориентированного подхода.
4. Спроектируйте программный продукт среднего размера с использованием спецификации требований к программному обеспечению, одной

из методологий проектирования (структурной или объектно-ориентированной) и соответствующей нотации.

5. Оцените программный проект с точки зрения повторного использования.
6. Примените ключевые элементы и типичные методы выявления и анализа требований для построения набора требований к программной системе среднего размера.
7. Опишите отличия между различными типами и уровнями тестирования для программных продуктов среднего размера.
8. Создайте, оцените и реализуйте тестовый план для модуля кода среднего размера.
9. Обсудите особенности и специфику тестирования объектно-ориентированного программного обеспечения.
10. Выберите (с обоснованием выбора) технологические подходы, наиболее подходящие для разработки и сопровождения нескольких существенно отличающихся программных продуктов.
11. Сравните жесткие и гибкие технологические подходы.
12. Опишите основные проблемы, связанные с эволюцией программного обеспечения и их влияние на жизненный цикл программного обеспечения.
13. Опишите процесс регрессионного тестирования и его роль в управлении выпуском версий продукта.
14. Оцените последствия запроса на изменение для существующего продукта среднего размера.
15. Обсудите преимущества и недостатки повторного использования программного обеспечения.
16. Продемонстрируйте основные принципы создания команды и управления коллективом на примере группового проекта.
17. Разработайте проектный план для программного проекта, включающий оценку размера и трудозатрат, график проекта, распределение ресурсов, управление конфигурацией, управление изменениями, а также выявление рисков, связанных с проектом и управление ими.
18. Сравните различные методы обеспечения качества программного продукта.
19. Объясните проблемы, возникающие на пути достижения очень высокого уровня надежности.

ЛИТЕРАТУРА

Если вы прочли эту книгу и поняли ее — обратитесь по адресу ... —
мы выплатим вам стоимость этой книги
и найдем высокооплачиваемую работу
не менее ... марок в месяц.

Надпись на карточке, приклеенной
в конце книги "Jetzt lerne ich Perl"

1. *Агафонов 1984* — Агафонов В. Н. Языки и средства спецификации программ (обзор) // Требования и спецификации в разработке программ. — М.: Мир, 1984, с. 285—344.
2. *Ахтырченко, Леонтьев 2001* — Ахтырченко К. В., Леонтьев В. В. Моделирование программной архитектуры // Вычислительные методы и программирование. 2001. Т. 2, с. 33—48.
3. *Баранов 1998* — Баранов С. Н. Управление программным проектом. Лекции по спецкурсу "Технология программирования". — СПб: Санкт-Петербургский государственный электротехнический университет, рукопись, 1998.
4. *Бек 2000* — Кент Бек. Экстремальное программирование // Открытые системы, № 1—2, 2000.
5. *Богатырев 2001* — Руслан Богатырев. Linux: истоки новой философии программирования // Мир ПК, № 1, 2001.
6. *Брукс 1999* — Брукс Ф. Мифический человеко-месяц или как создаются программные системы. — СПб.: Символ-Плюс, 1999.
7. *Буч 1998* — Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. — М.: Бином, СПб.: Невский диалект, 1998.
8. *Буч, Рамбо, Джекобсон 2000* — Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. — М.: ДМК, 1998.
9. *Вендров 2000* — Вендров А. М. Проектирование программного обеспечения экономических информационных систем. — М.: Финансы и статистика, 2000.
10. *Гамма, Хелм, Джонсон, Влиссидес 2001* — Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного программирования. Паттерны проектирования. — СПб.: Питер, 2001.
11. *Гейтс 2001* — Гейтс Б. Бизнес со скоростью мысли. — М.: ЭКСМО-Пресс, 2001.
12. *Гласс, Нуазо 1983* — Гласс Р., Нуазо Р. Сопровождение программного обеспечения. — М.: Мир, 1983.
13. *Грановская, Березная 1991* — Грановская Р. М., Березная И. Я. Интуиция и искусственный интеллект. — Л.: Изд-во ЛГУ, 1991.
14. *Деметрович, Кнут, Радо 1989* — Деметрович Я., Кнут Е., Радо П. Автоматизированные методы спецификации. — М.: Мир, 1989.
15. *Жоголев 1996* — Жоголев Е. А. Лекции по технологии программирования. — М.: МГУ, факультет ВМиК, рукопись, 1996.

16. *Йордон 2001* — Эдвард Йордон. Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте. — М.: ЛОРИ, 2001.

17. *Калянов 1996* — Калянов Г. Н. CASE структурный системный анализ (автоматизация и применение). — М.: ЛОРИ, 1996.

18. *Канер, Фолк, Нгуен 2000* — Сэм Канер, Джек Фолк, Енг Кек Нгуен. Тестирование программного обеспечения. — Киев: ДиаСофт, 2000.

19. *Керниган, Пайк 2001* — Керниган Брайан В., Пайк Роб. Практика программирования. — СПб.: Невский диалект, 2001.

20. *Кинг 1991* — Кинг Д. Создание эффективного программного обеспечения. — М.: Мир, 1991.

21. *Кнут 2000* — Дональд Кнут. Искусство программирования, том 1. Основные алгоритмы (3-е изд.). — М.: Издательский дом "Вильямс", 2000.

22. *Козлов 1999* — Козлов В. А. Открытые информационные системы. — М.: Финансы и статистика, 1999.

23. *Косовский, Хитров 1997* — Косовский Н. К., Хитров Д. В. Агрегация инноваций для повышения результативности систем // Первый российский философский конгресс. — СПб., 1997. Том V, с. 122—124.

24. *Кулаичев 1999* — Кулаичев А. П. Методы и средства анализа данных в среде Windows. — М.: Информатика и компьютеры, 1999.

25. *Липаев 1986* — Липаев В. В. Тестирование программ. — М.: Радио и связь, 1986.

26. *Майерс 1980* — Майерс Г. Надежность программного обеспечения. — М.: Мир, 1980.

27. *Орлов 2003* — Орлов С. Технологии разработки программного обеспечения. — СПб.: Питер, 2003.

28. *Пашкус, Мисько 1991* — Пашкус Ю. В. Мисько О. Н. Введение в бизнес. — Л.: "Северо-Запад", 1991.

29. *Пойа 1957* — Д. Пойа. Математика и правдоподобные рассуждения. — М.: Изд-во иностранной литературы, 1957.

30. *Поттосин 1997* — Поттосин И. В. "Хорошая программа": попытка точного определения понятия // Программирование, 1997, № 2, с. 3—17.

31. *Психология 2000* — Психология менеджмента: учебник. — СПб.: Изд-во Санкт-Петербургского университета, 2000.

32. *Романовский 1999* — Романовский И. В. Дискретный анализ. — СПб.: Невский диалект, 1999.

33. *Соммервилл 2002* — Соммервилл И. Инженерия программного обеспечения. — М.: Издательский дом "Вильямс", 2002.

34. *Тассел 1985* — Д. ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1985.

35. *Терехов, Туньон 1999* — Терехов А. А., Туньон В. Современные модели качества программного обеспечения // BYTE/Россия. — СПб.: "СК Пресс", № 12, 1999.

36. *Турский 1981* — Турский В. Методология программирования. — М.: Мир, 1981.

37. *Фаулер, Скотт 1999* — Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. — М.: Мир, 1999.

38. *Физики 1968* — Физики продолжают шутить. — М.: Мир, 1968.

39. *Филатова 1993* — Филатова Е. С. Соционика для Вас. — Новосибирск: "Сибирский хронограф", 1993.

40. *Филлипс, Гарсиа-Диас 1984* — Филлипс Д., Гарсиа-Диас А. Методы анализа сетей. — М.: Мир, 1984.

41. *Фокс 1985* — Дж. Фокс. Программное обеспечение и его разработка. — М.: Мир, 1985.

42. *Ходяченко 1999* — Ходяченко В. Б. Маркетинг: учебное пособие. — СПб.: Изд-во Санкт-Петербургского университета, 1999.

43. *Холстед 1981* — Холстед М. Х. Начала науки о программах. — М.: Финансы и статистика, 1981.

44. *Цейтин 1990* — Цейтин Г. С. На пути к сборочному программированию // Программирование, 1990, № 1, с. 78—92.

45. *Шевчук, Харахоркина, Журавлев, Армен 1998* — Шевчук О. Б., Харахоркина А. А., Журавлев Б. С., Армен В. А. Защищайтесь, сударь! Руководство по созданию и защите диссертаций. — СПб.: Политехника, 1998.

46. *Шлеер, Меллор 1993* — Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. — Киев: Диалектика, 1993.

47. *Abrial 1996* — Abrial J.-R. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

48. *Bach 1995* — James Bach. The Challenge of "Good Enough" Software // American Programmer, October 1995.

49. *Conger 1994* — Conger, Sue A. The New Software Engenering. Wadsworth Publishing Company, 1994.

50. *ISO/IEC 12207:1995* — ISO/IEC 12207:1995 Information Technology — Software Life Cycle Processes.

51. *ISO 9000 1992* — ISO 9000 International Standards for quality Management, Geneva, Switzerland, International Organization for Standartization, 1992.

52. *ISO 9001 1992* — Quality management and quality assurance standard — Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.

53. *Knuth 1992* — Donald E. Knuth. Literate Programming (CSLI Lecture Notes, No. 27). — Stanford: Center for the Study of Language and Information, 1992.

54. *Linger 1994* — Richard C. Linger. Cleanroom process model // IEEE Software, March 1994, pp. 50—58.

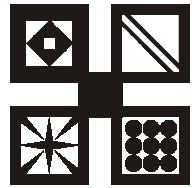
55. *Robillard, Robillard 2000* — Pierre N. Robillard, Martin P. Robillard. Types of collaborative work in software engineering // The Journal of Systems and Software 53, 2000, pp. 219—224.

56. *Safonov 1996* — Safonov V. O. Compiler Reengineering Methodology and its Application to SUN Pascal Compiler // Perspectives of System Informatics. Andrei Ershov Second International Memorial Conference. Akademgorodok, Novosibirsk, Russia, June 25—28, 1996, pp. 292—294.

57. *Shaw, Garlan 1996* — M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emergine Discipline. Prentice Hall, 1996.

58. *Thomsett 1990* — Rob Thomsett. Effective Project Teams: A Dilemma, a Model, a Solution. American Programmer, July-August 1990.

59. *Vliet 2000* — Vliet H. V. Software Engineering: Principles and Practice. Joht Wiley and Sons, 2000.



Глава 4

Языки программирования

Мы рассмотрим языки, участвующие в создании программы, поддерживающие основные технологические процессы. Для двух основных языковых групп — языков моделирования и языков программирования высокого уровня — можно выделить следующие черты, о которых также пойдет речь в этой главе.

- Язык обладает характеристиками.
- Язык содержит абстракции.
- Язык поддерживает методологию программирования.
- Язык принадлежит к семейству языков.
- Язык может быть специализированным для некоторой предметной области.

Завершим главу кратким обсуждением формальных и естественных языков.

4.1. Введение в языки программирования

> Основная задача языка программирования — помочь программисту в совершенствовании его искусства.
>
> *Чарльз Хоар*

С точки зрения программирования нас интересуют все те языки, которые участвуют в создании программы. На рис. 4.1 схематически указаны такие языки.

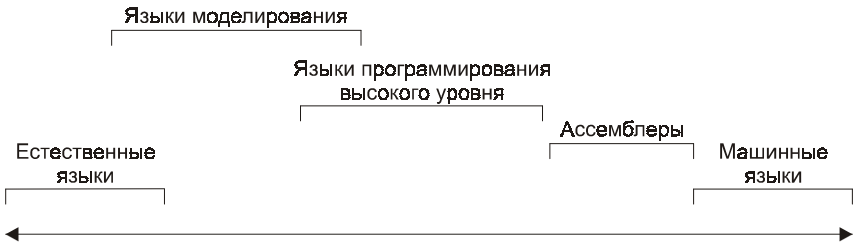


Рис. 4.1. Языки, участвующие в создании программы

Рассматривать язык программирования можно с нескольких позиций. Перечислим те из них, которые будут нам особенно полезны [Кауфман 1986].

- *Технологическая* позиция, отражающая взгляд человека, желающего или вынужденного пользоваться языком программирования как технологическим инструментом на каком-либо из этапов жизненного цикла программного обеспечения. Эта позиция использовалась в *гл. 3*.
- *Семиотическая* позиция, отражающая особенности языка как знаковой системы. Эта позиция доминирует в данной главе.
- *Математическая* позиция, представляющая математическую модель изучаемого языка.
- *Реализаторская* позиция, отражающая взгляд на язык реализаторов транслятора с этого языка и авторов документации. Эта позиция используется в *гл. 5*.
- *Авторская* позиция, отражающая позицию автора языка. Для понимания этой позиции следует обращаться к работам авторов языков программирования. Например, точка зрения Никлауса Вирта (Niclaus Wirth) отражена в статье [Вирт 1996].

4.1.1. Основные понятия и определения

4.1.1.1. Языки и знаки

Поцелуй может быть запятой, вопросительным знаком или восклицательным знаком.
Жанна-Мари Бужуа

Для информации, которой обмениваются люди посредством сообщений, в большинстве случаев имеются соглашения относительно их формы. Сообщения составляются на некотором языке с помощью знаков.

Введем простое определение знака [Лукин 1999]. *Форма* знака всегда материальна и передается через ощущения человека. Это может быть образ оптический (например, написанное слово), акустический (например, звук выстрела), обонятельный (например, запах свежего миндаля) или тактильный (например, поцелуй). Обычно нас интересует не форма знака, а то, о чем он сообщает — некий предмет, явление, процесс, идея. Замещая свою форму или представляя нечто отличное от нее, знак тем самым сообщает информацию. Заметим, что, говоря о представлении замещаемого предмета, мы обычно имеем в виду не сам предмет, а типичный образ, класс таких предметов. Этот представитель класса называется *денотатом*. Денотаты со своими формами связаны *значением* знака — отражением денотата в виде множества содержательных признаков, связывающего его с формой знака. Форма, значение и денотат образуют структуру знака, которую обычно изображают в виде "треугольника Фреге" (рис. 4.2).

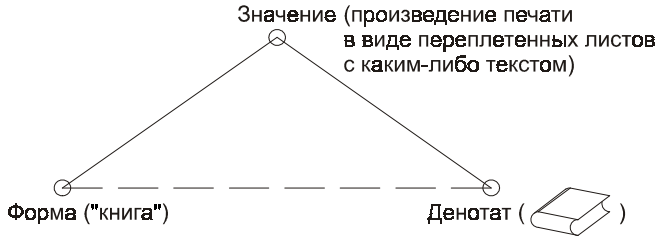


Рис. 4.2. Треугольник Фреге

Пунктирная линия, соединяющая форму и денотат, означает, что они соотносятся между собой не непосредственно, а через значение. Денотат отражается в значении, значение придает знаковость форме и вследствие этого форма представляет денотат.

Семиотика — наука, изучающая языковые знаковые системы.

Модель передачи сообщений

Одна из основных задач, стоящих перед программистом, — добиться взаимопонимания с компьютером. Чтобы разобраться в возникающих при этом проблемах, рассмотрим модель передачи сообщений [Кауфман 1986] (рис. 4.3).

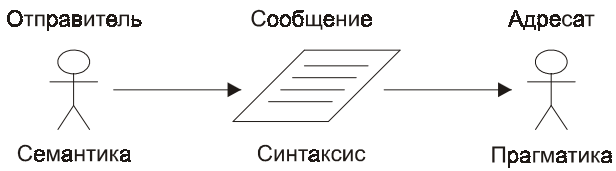


Рис. 4.3. Модель передачи сообщений

В этой модели присутствуют два основных действующих лица.

- Отправитель (он же автор, генератор сообщения).
- Адресат (он же получатель, читатель, слушатель сообщения).

Собственно сообщение представляется последовательностью знаков (например, текст или ряд звуков). Его смысл — нечто, обозначаемое сообщением, в соответствии с правилами, известными и отправителю, и адресату.

Укажем, что изучают части науки семиотики, которые фигурируют в этой модели.

- *Синтаксис* — правила построения допустимых сообщений.
- *Семантика* — правила сопоставления смысла таким сообщениям.
- *Прагматика* — правила, регулирующие использование сообщений. Обычно прагматика ассоциируется с целями пишущего, т. е. с желаемым, а не фактическим поведением (использованием сообщения) адресата.

Известны языки разговорный и письменный, язык жестов и мимики, язык для слепых, воспринимаемый осязанием.

> ***О языке влюбленных***
>
> Отличительными признаками диалога влюбленных являются: рыхлый синтаксис, могучая прагматика и возможное полное отсутствие семантики.

С помощью приведенной модели передачи сообщений можно классифицировать возникающие при этом недоразумения [Кауфман 1986].

- *Синтаксические недоразумения.* Отправитель может подразумевать одну структуру сообщения, а адресат — другую. Примером может являться классический королевский указ — "Казнить нельзя помиловать!". Кстати, добрый текстовый редактор (Microsoft Word), с помощью которого создавалась книга, при анализе этой фразы выдает рекомендацию "Возможно, не хватает запятой после слова “нельзя”".
- *Семантические недоразумения.* Отправитель и адресат могут по-разному понимать некоторые слова. Пример — фраза на пакетике риса "варить до готовности".
- *Прагматические недоразумения.* Адресат может придать сообщению отправителя совершенно другую интерпретацию. Примером является огромное количество анекдотов, посвященных так называемой "женской" логике. Конечно, особенности "женской" логики целиком лежат в сфере семантики, хотя проявляются они в ответных сообщениях, диктуемых прагматикой получательницы.

> ***О деформации знака в случае затрудненного восприятия***
>
> Получатель знака может деформировать его структуру, превращая один вид знака в другой. Это типично для ситуаций, когда имеет место восприятие, затрудненное по тем или иным причинам. В таких ситуациях получатель стремится свести непонятное к уже знакомому и ясному для него [Лукин 1999]: "Куприна, уже подвыпившего, раз привели к Балтрушайтису, чтобы представить: "Знакомьтесь: Куприн, Балтрушайтис". Куприн же: "Спасибо: уже балтрушался". Ему показалось спьяну глагол "балтрушайтесь" — в значении понятном весьма: "Угощайтесь".

Добавим, что при анализе сообщений на естественном языке могут возникнуть многие другие проблемы. Например, при омонимии у одного знака могут быть несколько пар значение-денотат. Хрестоматийный пример — "закрой замок на замок, чтобы замок не замок".

Языковые знаковые системы

Язык вообще — это определенный способ сопоставления объектам `Ri`, которые рассматриваются как некоторая первичная реальность, объектов `Li`, называемых именами `Ri` и рассматриваемых как нечто вторичное, специально созданное для сопоставления объектам `Ri` [Турчин 2000]. По отношению

к своему имени `Li` объект `Ri` называют его *значением*. Совокупность всех объектов `Li` также часто называют языком.

Отметим две основные функции языка.

- *Коммуникативная* функция. Это первая функция языка, поскольку он возникает как средство связи.
- *Моделирующая* функция. Язык используется в качестве средства построения моделей действительности.

Интересное и образное определение языка принадлежит Тузову [Тузов 1990]. *Язык* — средство описания информационных процессов. Предложения языка и текст — застывшие формы процессов. Они превращаются в активный процесс во время выполнения его человеком или вычислительной машиной.

Очень часто *языком* называют совокупность синтаксически правильных и семантически интерпретируемых (хотя бы частично) текстов.

Для человека естественно линейное и последовательное построение предложений языка. Однако существуют и более сложные способы построения предложений. Например, марсианская грамматика (**http://progstone.nm.ru**) имеет свои правила и использует Северный голос для эмоций, Южный голос для действия, Восточный голос для речи и Западный голос для обстоятельств. На Марсе есть марсиане, едящие мышей и мыши, скрывающиеся среди скал и едящие скалы. Марсиане записывают свою литературу на больших скалах с помощью маленьких обломков скал. Наиболее близкий марсианский аналог нашего "Преступления и наказания" (написанного в 1866 году Ф. М. Достоевским [Достоевский 1985]) называется "Кто украл мою мышь?" (табл. 4.1).

Таблица 4.1. *Фрагмент марсианской литературы*

Эмоция	Действия	Речь	Обстоятельства
Сердитый	Подкрадывается	Ужасный марсианин	Холодный, мрачный
Определенно	Бить	Умри! Умри! Умри!	Старая пещера
Стыдно	Украсть мышь		Мертвый марсианин

Неожиданный пробел в Восточном голосе — непроизнесенное проклятие, очень реальное само по себе. Обратим внимание на то, что линейное использование последовательностей символических преобразований, хорошо понятное человеку, становится непонятным марсианам, мозг которых позволяет почувствовать сложные отношения между независимыми действиями.

Об универсальном языке

В мире существует огромное количество языков: и естественных, и моделирования, и программирования. Известно, что в начале XX века был изобретен универсальный естественный язык эсперанто. Он был специально разработан так, чтобы облегчить его изучение. Конец этой истории известен — в наше время эсперанто знают единицы, а тысячи языков и диалектов звучат и не собираются исчезать. В мире языков программирования почти все языки задумывались как универсальные. Наиболее известными были следующие попытки разработать и внедрить универсальный язык:

- язык PL/1 был создан в компании IBM в конце 60-х годов XX века;
- язык Algol-68 являлся итогом многолетней работы (1963—1968 гг.), проведенной Международной Федерацией по обработке информации;
- язык Ada был создан по заказу министерства обороны США в конце 70-х — начале 80-х годов XX века.

Ни одна из этих попыток не достигла успеха. Объяснить эту ситуацию можно с помощью гипотезы Хурфа-Сэпира, утверждающей, "что наш язык, наш словарь, словесные формы, контролируют и ограничивают мышление, обусловливая нашу способность мыслить" [Васкевич 1996].

То, что мы произносим, — это то, о чем мы думаем. Пусть в языке какого-то народа отсутствуют, например, слова, отражающие само понятие времени (или понятия снега, пустынь и т. п.). Даже если этот народ перейдет к другому языку, в котором такие слова есть, он не сможет их использовать, поскольку не умеет думать об этих понятиях.

Необходимость существования многих языков и систем обозначений порождается принципом расслоения абстракции. Человек не может напрямую работать со сложными системами, одновременно отслеживать множество взаимосвязанных элементов. Слои (уровни) абстракции позволяют людям понизить сложность систем в мире, математике или программировании до управляемых размеров.

Формализация языка

Формализованный язык можно определить следующим образом [Турчин 2000]. Рассмотрим двухэтажную языковую модель действительности (рис. 4.4). Ситуация `Si` кодируется языковым объектом `Li`. Объект `L1` есть имя для `S1`. Некоторое время спустя ситуация `S1` сменяется ситуацией `S2`. Осуществляя некоторую языковую деятельность, преобразуем `L1` в другой объект — `L2`. Если наша модель правильна, то `L2` есть имя `S2`. В результате, не зная реальной ситуации `S2`, мы можем получить представление о ней путем декодирования языкового объекта `L2`. Выполнение преобразования `L1->L2` определяет, будет ли язык формализованным.

- Для *формализованного языка* преобразование `L1->L2` определяется исключительно языковыми объектами `Li`, которые участвуют в нем и не зависят от языковых представлений `Si`, соответствующих им по семантике языка.
- Для *неформализованного языка* результат преобразования языкового объекта `Li` зависит не только от вида самого представления `Li`, но и от представления `Si`, которое он порождает в голове человека, от ассоциаций, в которые он входит.

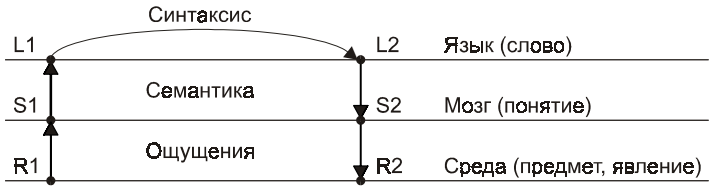


Рис. 4.4. Двухэтажная языковая модель действительности

О железнодорожном языке

"Платформу Красные Зори поезд проследует без остановки". Обратим внимание, что машинист употребил существительное "остановка", а не глагол "останавливаться". Остановка — очень важное для железнодорожников понятие. Поезд может "остановиться", но не "иметь остановки". Турчин [Турчин 2000], приводя подобный пример, указывает на формализацию языка, употребляемого в узких профессиональных целях.

Человек способен воспринимать самые неформализованные языки. А компьютер не понимает, точнее, не может исполнить программу на неформальном языке. Именно поэтому важное место в изучении программирования всегда занимают формальные алгоритмические языки программирования.

О формализации неформализованного

Формализация неформализованного — процесс неформализуемый. Хотя с этим пытаются бороться логики и военные.

О формуле любви

Формула любви не поддается формализации. В лучшем случае она может быть представлена только в виде весьма грубой модели (**http://asu.pstu.ac.ru/book/intelekt/index.html**).

4.1.1.2. Языки моделирования

Язык моделирования — набор правил, определяющих построение моделей (упрощенного представления реальности), включающий их визуализацию и определение структуры и поведения. Язык моделирования включает:

- *элементы модели* — фундаментальные концепции моделирования и их семантику;

- *нотацию* — визуальное представление элементов моделирования;
- *руководство по использованию* — правила применения элементов в рамках построения моделей предметной области.

4.1.1.3. Языки программирования высокого уровня

Язык программирования (алгоритмический язык) — набор правил, определяющих, какие последовательности символов составляют программу (синтаксические правила) и какие вычисления описывает программа (семантические правила).

Программа — текст, задающий множество процессов вычислений, в соответствии с которым исполнитель, понимающий программу, разворачивает какой-то один из них.

Выделим три основные *характеристики языков программирования* [Калинин, Мацкевич 1991].

- *Уровень* языка характеризуется сложностью задач, решаемых с помощью этого языка.

> ***О том, чем определяется уровень языка программирования***
>
> Интуитивно любой программист отличит язык программирования высокого уровня от языка программирования низкого уровня. В чем же состоит различие? Чем определяется уровень? Программирование представляет собой отображение в программах объектов, понятий и явлений предметной области задачи. Чем более адекватно можно выполнить это отображение, тем выше уровень языка программирования. А отображение будет выполнено тем лучше, чем богаче возможности типообразования языка программирования.

- *Мощность* языка характеризуется количеством и многообразием задач, алгоритмы решений которых можно записать, используя этот язык.
- *Концептуальная целостность* характеризуется свойствами совокупности понятий, служащих для описания этого языка, и включает три взаимосвязанных аспекта.
  - *Экономия понятий.* Язык должен достигать своей максимальной мощности минимальным количеством понятий.
  - *Ортогональность понятий.* Между понятиями не должно быть взаимного влияния. Если понятие используется в различных контекстах, то правило его использования должно быть одним и тем же.

> ***О неортогональности понятий даже самых лучших языков***
>
> К сожалению, даже такие языки, как Pascal, допускают неортогональные конструкции. Например, пользователь может определить процедуры только с фиксированным числом параметров, однако некоторые стандартные процедуры (например, `writeln()`) могут быть вызваны с переменным числом параметров.

  - *Единообразие понятий.* Это требование согласованного единого подхода к описанию и использованию всех понятий.

Существует большое количество других характеристик.

- *Надежность* языка должна обеспечивать минимум ошибок при написании программ. Более того, язык должен быть таким, чтобы неправильные программы было трудно писать. Обратим внимание на то, что достаточно сложно затруднить написание неправильных программ, не внося ненужных трудностей в составление правильных.
- *Удобочитаемость* — легкость восприятия программ человеком. Это характеристика важна при коллективной работе, когда несколько человек работают с одними и теми же текстами программ.
- *Полнота* характеризует способность описать класс задач в некоторой предметной области.
- *Гибкость* характеризует легкость выражения необходимых действий.

Отметим два свойства, которые не влияют на процесс разработки, но играют важную роль, отражая реализаторскую позицию.

- *Мобильность* — независимость от аппаратных средств, обеспечивающая переносимость программного обеспечения.
- *Эффективность реализации* должна включать не только эффективную реализацию самого компилятора, но и эффективную генерацию компилятором кода программ.

И это только некоторые из свойств и характеристик. В одном из докладов военно-морскому флоту [Фокс 1985] исследователи насчитали более 2570 различных возможностей или свойств, которые можно выделить в языках программирования.

> ***О конфликтовании характеристик языков***
>
> Следует заметить, что многие из перечисленных характеристик конфликтуют друг с другом. Например, надежность и эффективность. Это подчеркивает сложности, с которыми приходится сталкиваться создателям новых языков.

4.1.2. История и эволюция

4.1.2.1. История и эволюция языков моделирования

Покрывало алтаря в одной эпохе — это коврик для ног в следующей.
Марк Твен

Принято выделять всего два поколения языков моделирования.

- Первое поколение — *языки структурного системного анализа и проектирования*. Появились в середине 70-х годов XX века.
- Второе поколение — *объектно-ориентированные языки моделирования*. Первые языки появились в конце 70-х годов XX века, однако их массовое появление и развитие приходится на начало 90-х годов этого века.

Укажем некоторые важные даты, связанные с языками моделирования.

- 1976 г. — Питером Ченом (Peter Pin Shan Chen) разработана диаграмма сущность-связь.
- 1995 г. — Консорциум OMG признает целесообразность поиска индустриальных стандартов в области языков моделирования.
- 1996 г. — Появление первых документов, содержащих описание языка UML.
- 1997 г. — Публикация документа с описанием языка UML версии 1.0.

4.1.2.2. История и эволюция языков программирования

К сегодняшнему дню насчитывают шесть поколений языков программирования. Каждое из последующих поколений по своей функциональной мощности качественно отличается от предыдущего.

- Первое поколение — *машинные языки*. Появились в середине 40-х годов XX века.
- Второе поколение — *ассемблеры*. Фактически это те же машинные языки, но более красиво "обернутые". Появились в конце 50-х годов XX века. Кстати, первые версии языка Fortran можно охарактеризовать как язык верхнего ассемблерного уровня.
- Третье поколение — *процедурные языки*. Появились в начале 60-х годов XX века. К этому поколению относят универсальные языки высокого уровня, с помощью которых можно решать задачи из любых областей (например, Algol-60).
- Четвертое поколение — *языки поддержки сложных структур данных* (например, SQL). Появились в конце 60-х годов XX века.
- Пятое поколение — *языки искусственного интеллекта* (например, Prolog). Появились в начале 70-х годов XX века.
- Шестое поколение — *языки нейронных сетей* (самообучающиеся языки). Основные исследовательские работы в этой области начались в середине 80-х годов XX века. А. И. Галушкин относит возникновение российской научной школы в теории нейронных сетей к концу 1960-х годов.

Даты создания многих языков программирования будут указаны *в разд. 4.5*. Здесь мы приведем лишь языки, которые считаются первыми:

- 1945 г. — Конрадом Цузе (Conrad Zuse) разработан первый язык программирования Plankalkuel;
- 1949 г. — разработанный Джоном Мочли (John Mauchly) язык Short Code американцы считают первым языком программирования высокого уровня.

4.1.3. Классификация языков

4.1.3.1. Классификация языков моделирования

Языки моделирования могут быть разделены на три группы:

- языки *описания архитектур* (Architecture Description Language), предоставляющие средства для моделирования концептуальной программной архитектуры. Их также называют языками проектирования "в большом". Основными элементами таких языков, как правило, являются:
  - *компоненты*, для которых могут быть определены поддерживаемые ими интерфейсы (порты);
  - *соединители*, реализующие протоколы взаимодействия компонентов и также поддерживающие определенные интерфейсы (роли);
  - *архитектурные конфигурации*, которые являются композицией компонентов и соединителей и могут быть представлены в виде иерархической структуры;
  - *ограничения* на композицию компонентов и соединителей в рамках архитектурных конфигураций;
- языки *проектирования модулей* (языки проектирования "в малом");
- языки спецификаций.

Отдельно следует выделить еще две группы языков моделирования, имеющих большое значение для многих областей программирования:

- языки моделирования данных *(см. разд. 4.4.3)*;
- языки моделирования знаний *(см. разд. 4.4.4)*.

Эти две группы являются вспомогательными языками с точки зрения первых трех групп. Например, некоторый язык моделирования данных может быть составной частью некоторого языка спецификаций.

4.1.3.2. Классификации языков программирования

В настоящее время сложилось довольно много классификаций языков программирования. Далее мы приведем самые известные из них.

Классификация по поддерживаемым методологиям

Классификация языков по поддерживаемым методологиям появилась примерно в 80-х годах XX века. Мы выделили следующие основные группы языков:

- языки императивного программирования *(см. разд. 2.2.1)* и две важнейшие подгруппы:
  - языки структурного императивного программирования *(см. разд. 2.3.1)*. Эти языки более известны под кратким именем — языки структурного программирования;

- языки императивного параллельного программирования *(см. разд. 2.4.1)*. Эти языки также обычно называют кратко — языки параллельного программирования;

- ❐ языки объектно-ориентированного программирования *(см. разд. 2.2.2)*;
- ❐ языки функционального программирования *(см. разд. 2.2.3)*;
- ❐ языки логического программирования *(см. разд. 2.2.4)*;
- ❐ языки программирования в ограничениях *(см. разд. 2.2.5)*.

Классификация по принадлежности к семействам

Основная задача классификации языков по принадлежности к семействам — проследить их родственные взаимосвязи (фактически, построить генеалогическое дерево) с целью выяснения их влияния друг на друга и, следовательно, на характеристики и свойства языков. Это нечеткая классификация, которая может вызвать спорное отношение к ней. Выделим десять основных семейств.

- ❐ Семейство универсальных языков *(см. разд. 4.5.1.1)*.
- ❐ Семейство уникальных языков *(см. разд. 4.5.1.2)*.
- ❐ Семейство языков параллельного программирования *(см. разд. 4.5.1.3)*.
- ❐ C-семейство языков *(см. разд. 4.5.1.4)*.
- ❐ Pascal-семейство языков *(см. разд. 4.5.1.5)*.
- ❐ Ada-семейство языков *(см. разд. 4.5.1.6)*.
- ❐ Simula-семейство языков *(см. разд. 4.5.1.7)*.
- ❐ Lisp-семейство языков *(см. разд. 4.5.1.8)*.
- ❐ Prolog-семейство языков *(см. разд. 4.5.1.9)*.
- ❐ Forth-семейство языков *(см. разд. 4.5.1.10)*.

Классификация по ориентации на предметные области

Язык программирования может быть специализированным для некоторой предметной области. Выделим девять таких областей, хотя конечно их гораздо больше.

- ❐ Языки форматирования текстов *(см. разд. 4.5.2.1)*.
- ❐ Языки разметки *(см. разд. 4.5.2.2)*.
- ❐ Языки сценариев (скриптов) *(см. разд. 4.5.2.3)*.
- ❐ Промежуточные языки программирования *(см. разд. 4.5.2.4)*.
- ❐ Языки программирования агентов *(см. разд. 4.5.2.5)*.
- ❐ Языки создания графики *(см. разд. 4.5.2.6)*.
- ❐ Языки описания аппаратуры *(см. разд. 4.5.2.7)*.

- Языки описания виртуальной реальности *(см. разд. 4.5.2.8)*.
- Языки конфигурирования *(см. разд. 4.5.2.9)*.

Классификация по степени абстракции от аппаратуры

По степени абстракции от машины можно выделить три группы языков.

- *Языки низкого уровня*. Такие языки имеют простые машиноподобные команды и осуществляют прямой доступ к памяти. Пример — ассемблер для любой архитектуры.
- *Языки высокого уровня*. Языки предоставляют возможность определять сложные структуры данных, доступ к памяти осуществляется через операции. Примерами языков этого уровня являются Pascal, C и Ada.
- *Языки сверхвысокого уровня*. Команды исполняются на полностью абстрактной машине, почти полностью скрыт доступ к памяти. Пример таких языков — Prolog, SETL, APL, Miranda.

С точки зрения порождаемого кода выделяют две ветви в развитии языков.

- Языки (как правило, компилируемые), ориентированные на скорость исполнения кода программы.
- Языки (и компилируемые, и интерпретируемые), ориентированные на высокий уровень и удобство программирования.

4.2. Языки, поддерживающие классические технологические процессы

4.2.1. Возникновение и исследование идеи

Когда слепой жук ползет по поверхности шара, он не замечает,
что пройденный им путь изогнут, мне же посчастливилось заметить это.
Альберт Эйнштейн (объясняя своему девятилетнему сыну, чем он прославился в науке)

Процесс возникновения и исследования идеи является творческим и неформализуемым. Он наиболее адекватно описывается естественным языком.

4.2.2. Управление

Языки управления в большинстве случаев — языки графовых структур. *Сетевая модель проекта* представляет собой графическое описание плана, показывающее взаимосвязь между всеми заданиями, которые надо выполнить для завершения проекта [Филлипс, Гарсиа, Диас 1984]. Сеть состоит из ориентированных дуг или отрезков, соединяющих узлы попарно. Элементы сети, характеризуемые затратами времени (дуги или отрезки), называются

работами. Узлы, обозначаемые кружками или прямоугольниками, представляют *события*. События являются точно заданными моментами времени. Направление дуги или отрезка определяет отношение предшествования. На отрезке сети (рис 4.5) событие `i` должно произойти до начала работы `A`, а событие `j` не может произойти до завершения работы `A`.

Рис. 4.5. Отношение предшествования на отрезке сети

Иногда отношение предшествования между работами нельзя представить точно с помощью обычной структуры работ и событий. Пусть нам нужно представить на схеме изображение следующей последовательности:

- работа `D` следует за работами `A` и `B`;
- работа `C` следует за работой `A` (но не за работой `B`).

Схема, приведенная на рис. 4.6, *а*, является неправильной, поскольку показывает, что работа `C` следует за работами `A` и `B`. Для правильного представления изображения следует ввести фиктивную работу (обозначим ее `X` на рис. 4.6, *б*), продолжительность которой равна нулю.

Рис. 4.6. Неправильная структура работ (*а*) и введение фиктивной работы (*б*)

Модель сетевого графика имеет большое значение как хороший организующий инструмент. Выделение критических путей позволяет организаторам выделить область наибольшего внимания, хотя по многим причинам она не исчерпывается критическими путями. Полезно несколько расширять область контроля, вводя в нее субкритические пути, которые имеют малый резерв времени, и небольшое изменение продолжительности может вывести их на критический путь [Романовский 1999].

О первом языке управления

Иронически можно заметить, что исторически первым языком управления на Руси был "русский матерный".

4.2.3. Анализ требований и проектирование

Подробно языки моделирования, применяемые при ведении процессов анализа требований и проектирования, будут рассмотрены в *разд. 4.4.*

4.2.4. Программирование (реализация)

Подробно языки программирования высокого уровня, применяемые при ведении процесса программирования, будут рассмотрены в *разд. 4.5.*

4.2.5. Тестирование и отладка

Чем реже встречаются ошибки, тем они ценнее.
Станислав Ежи Лец

В процессе тестирования и отладки применяется не так много специализированных языков. Можно отметить язык сценариев для тестового монитора, использующийся на этапе тестирования. На этапе отладки используется язык команд отладчика.

4.2.6. Ввод в действие

Как правило, единственный язык, который используется в этом процессе, — язык инсталляционных (устанавливающих) программ. Инсталляционные программы задают пользователю вопросы и сообщают об устанавливаемой конфигурации на подмножестве естественного языка.

Некоторые инсталляторы имеют богатые языки сценариев формирования дистрибутива, но знать их необязательно, поскольку над ними практически всегда существует графический интерфейс.

4.2.7. Эксплуатация и сопровождение

Сопровождение программы предполагает внесение изменений на том же языке программирования, на котором была написана программа. Исключение составляют реинжиниринг и программирование заново (с использованием метода обратной инженерии). В этом случае программа или ее существенная часть может быть:

- преобразована к программе на другом (более современном) языке с помощью конвертора (рис. 4.7, пунктирная линия). Проблемы и трудности, возникающие при конвертации программ с языков COBOL и PL/1, изложены в работе [Terekhov, Verhoef 2000];
- преобразована с помощью восстановления спецификации к программе на языке моделирования, с которого потом уже будет получена программа на современном языке программирования (рис. 4.7, сплошная линия).

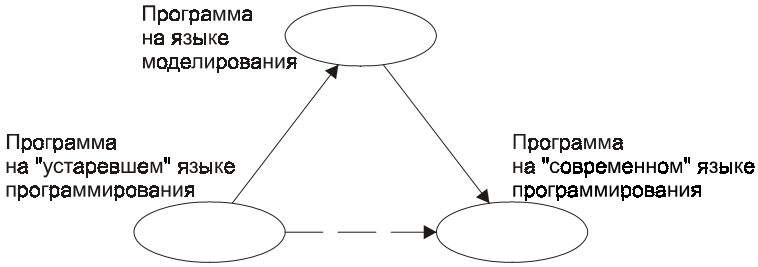


Рис. 4.7. Конвертирование программы на современный язык программирования

4.2.8. Завершение эксплуатации

Водой небытия зародыш мой вспоен,
огнем страдания мой мрачный дух зажжен,
как ветер, я несусь из края в край вселенной
и горсточкой земли окончу жизни сон.

Омар Хайям

Завершение эксплуатации анонсируется на естественном языке. Программа, начиная с некоторой версии, будет выдавать пользователю предупреждение о прекращении сопровождения.

4.3. Языковые абстракции

4.3.1. Отступление "об абстрагировании"

Сотри случайные черты, и ты увидишь — мир прекрасен.

Александр Блок

Абстрагирование — принцип игнорирования второстепенных аспектов предмета с целью выделения главных. Математики считают, что абстрактные системы помогают нам организовать наш мир и наше мышление.

Абстрагирование в информатике имеет некоторые особенности, отмеченные Питером Деннингом (Peter Denning) в следующем определении. *Абстрагирование в информатике* — это моделирование возможных реализаций. Модели подавляют детали, но оставляют существенные особенности. Отличие от математики еще и в том, что мы рассматриваем абстракции одновременно как по отношению к решаемой проблеме, так и по отношению к физической машине (второе в математических абстракциях отсутствует). Мы рассмотрим три типа абстракций в языках программирования [Appleby 1991].

- Абстракцию данных *(см. разд. 4.3.2)*.
- Абстракцию управления *(см. разд. 4.3.3)*.
- Абстракцию модульности *(см. разд. 4.3.4)*.

Абстрагирование особенно актуально для двух групп языков, которые мы определили как языки моделирования и программирования.

4.3.2. Абстракция данных

4.3.2.1. Данные и типы данных

Исключительное использование сильно типизированного языка является фактором, в наибольшей степени определяющим возможность проектирования сложной системы в короткий срок.

Никлаус Вирт

Данные — это отдельные элементы, которые могут быть собраны вместе некоторым образом. Большинство языков программирования имеют хотя бы один встроенный тип данных (как правило, целые числа).

Исторически можно проследить несколько этапов в становлении типов данных в языках программирования.

- Первый этап — в описании языка заданы все типы данных, которые могут быть использованы. Например, для языка FORTRAN II это пять простых типов: целые, вещественные, вещественные двойной точности, комплексные и логические. Единственный структурный тип — массив.
- Второй этап — появление в языках разветвленных средств конструирования типов данных. Этот этап характеризуют языки Algol-68 и Pascal.
- Третий этап — осознание необходимости абстрактных типов данных. Языки программирования — CLU, Modula, Ada.
- Четвертый этап — появление понятия класса в языках программирования. Языки программирования — Smalltalk, Object Pascal, C++.

4.3.2.2. Эволюция определения типа данных

Современное определение типа данных звучит так. *Тип данных* — это множество значений, определяемых посредством множества операций.

Далее мы перечислим еще три определения типа данных, появлявшихся последовательно [Замулин 1990]. В них прослеживается изменение математической позиции с теоретико-множественной (которая допускает готовые актуально-бесконечные множества) на конструктивную. Конструктивная позиция рассматривает только потенциально-бесконечные множества, конечные в каждый момент времени, но дающие возможность добавлять новые элементы.

- Тип данных — множество значений. Это самое первое определение типа данных, просуществовавшее более 15 лет. Оно было актуально для того времени, когда языки программирования предоставляли лишь ограниченное количество встроенных стандартных типов данных. Неестественность

этого определения для конструктивного по своей сути программирования была отмечена в 1973 году Джеймсом Моррисом (James Morris) в знаменитой статье "Типы — не множества" [Morris 1973].

- Тип данных характеризуется не столько множеством значений, сколько множеством операций. Таким образом, для определения типа данных требуется образовать множество операций, конструирующих и анализирующих его значения. Это одно из первых конструктивных определений.
- Тип данных — множество операций, задающих интерпретацию значений универсального пространства значений. Это конструктивно-операционное определение типа данных.

Об основном вопросе в области типов данных

Основной вопрос в области типов данных можно сформулировать так. Что первично — данные или типы данных? Мы будем становиться на точку зрения конструктивистов (рассматриваются только потенциально-бесконечные множества), которая ближе программированию. Первичность типов данных подчеркивает необходимость двух классов операций: конструкторов данных определенного типа и анализаторов, определяющих принадлежность данных к определенному типу.

4.3.2.3. Абстрактные типы данных

Абстрактным типом данных будем называть модуль, внутренняя часть которого скрыта от пользователя, и работа с модулем разрешается только при помощи операций.

Развернутое определение абстрактного типа данных включает в себя следующие части:

- внешность, содержащую имя типа данных и операций, с указанием типов их аргументов и типа значения;
- абстрактное описание операций и объектов, с которыми они работают, средствами некоторого языка спецификаций;
- конкретное описание операций и объектов средствами обычного языка программирования;
- описание того, насколько абстрактное описание представляется конкретным.

Обоснование применимости концепции абстрактных типов данных в практике программирования было сделано Чарльзом Хоаром (Charles Antony Richard Hoare). Некоторая программа, работающая с данными типа `T` и содержащая последовательность операторов, реализующих операции с этим типом, может быть преобразована в функционально-эквивалентную программу. В результирующей программе каждая операция с типом `T` может быть описана в виде функции, а все явно запрограммированные действия с этим типом могут быть заменены вызовами соответствующих функций.

4.3.2.4. Разновидности полиморфизма

Все языки могут быть разделены на две группы.

- *Мономорфные* языки, в которых каждое значение и каждая переменная интерпретируются как принадлежащие одному и только одному типу. Хоар еще в 1975 году отмечал, что размышлять о типах легче, если придерживаться следующего важного принципа: каждое значение, переменная и выражение принадлежат одному и только одному типу.
- *Полиморфные языки*, в которых некоторые значения и переменные интерпретируются как принадлежащие более чем одному типу.

Полиморфные типы — это типы, чьи операции применимы к значениям более чем одного типа. *Полиморфные функции* — функции, чьи операции (параметры) могут иметь значения более одного типа.

Выделяют два класса и четыре основных подкласса *полиморфизма* [Cardelli, Wegner 1985].

- *Универсальный* полиморфизм, который будет нормально работать с бесконечным количеством типов.
  - *Параметрический* полиморфизм, в котором полиморфная функция имеет явный или неявный параметр — тип, определяющий тип аргумента для каждого приложения функции. Примером могут служить родовые функции.
  - Полиморфизм *включения*, в котором объект можно рассматривать как принадлежащий многим классам. Полиморфизм включения позволяет моделировать подтипы и наследование.

  О подклассах универсального полиморфизма

  В некотором смысле параметрический полиморфизм аналогичен связыванию на этапе компиляции, а полиморфизм включения — связыванию на этапе исполнения.

- *Неуниверсальный* полиморфизм, работающий на конечном множестве различных и потенциально не связанных друг с другом типов.
  - *Перегрузка* — это использование для различных функций одного и того же имени, причем для определения требуемой функции используется контекст.
  - *Приведение* — это семантическая операция, необходимая для выполнения преобразования аргумента к типу, требуемому функцией.

Поясним разницу между перегрузкой и приведением на простом примере. Рассмотрим оператор сложения, применяемый к различным типам операндов:

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

Неуниверсальный полиморфизм оператора сложения + может быть представлен тремя случаями.

- Оператор имеет четыре перегруженных смысла, по одному для каждой комбинации.
- Оператор имеет два перегруженных смысла, для целых и вещественных. В том случае, когда один из операндов целый, а другой вещественный, целый будет приведен к вещественному.
- Оператор определен только для вещественных аргументов. Целочисленные аргументы всегда приводятся к вещественным.

> ***О перегрузке***
>
> Бьерн Страуструп (Bjarne Stroustrup) в статье "Generalizing Overloading for C++2000" (**http://www.research.att.com/~bs/papers.html**), датируемой 1 апреля 1998 г., предлагает развивать идеи перегрузки в языке C++. Например, предлагается перегружать не только пробелы, но и пропущенные пробелы.

4.3.2.5. Статический и динамический контроль типов

Контроль типов должен определять возможность применения данной операции к данным аргументам.

Существуют два типа контроля.

- *Статический контроль* — контроль периода компиляции. Статический контроль обеспечивает эффективную реализацию полиморфизма имен операций.
- *Динамический контроль* — контроль периода выполнения. Он обеспечивает универсальность и надежность, а также дополнительную гибкость.

4.3.2.6. Статически и динамически типизируемые языки программирования

Языки программирования могут быть статически и динамически типизируемыми.

- В *статически типизируемых* языках программирования семантика каждой языковой конструкции определяется по тексту программы, а не во время выполнения. В описании переменной всегда указывается тип ее возможных значений.
- В *динамически типизируемых* языках семантика конструкции определяется во время исполнения.

На рис. 4.8, *а* и *б* указана связь имени объекта (переменной) с типом и значением для статических и динамических языков программирования, соответственно. Одинарная линия на рисунке показывает переменную связь, а двойная линия — фиксированную, жесткую связь.

Рис. 4.8. Связь переменной с типом в статически (*а*) и динамически (*б*) типизируемых языках

Одним из наиболее интересных динамически типизируемых языков программирования является язык "Автокод Эльбрус" [Сафонов 1989]. При определении переменной целочисленного типа данных отводится лишь область памяти необходимого формата. Например, описание `ф64 a` указывает на необходимость отведения для переменной `a` 64 битов, в которых далее может быть размещено целое число. Динамическая типизация может быть эффективно поддержана аппаратной реализацией (например, архитектура Эльбрус для языка "Автокод Эльбрус").

Статически типизируемые языки программирования чаще всего имеют смешанный контроль типов данных. Можно выделить три основные группы таких языков.

- *Слабо типизируемые* языки, в которых информация о типе служит только для обеспечения корректности программы на машинном уровне (например, язык C).
- *Сильно типизируемые* языки, осуществляющие полный контроль типов (например, язык Ada).
- *Строго типизируемые* языки. В таких языках (например, в языке Pascal) программы, содержащие конструкции с возможной типовой ошибкой, считаются недопустимыми, даже если ошибка никогда не может возникнуть.

4.3.3. Абстракция управления

4.3.3.1. Структурное программирование

Изучение абстракции управления мы начнем со структурного программирования. Самое общеизвестное определение *структурного программирования* — это подход к программированию, в котором для передачи управления в программе используется три конструкции: последовательная, выбора и итеративная.

Примерно так выглядело историческое становление концепции структурного программирования:

- классическая теорема Боэма и Джакопини о структурном программировании утверждает, что всякую правильную программу (т. е. программу

с одним входом и одним выходом, без зацикливаний и недостижимых веток) можно записать с использованием следующих логических структур:

- последовательности двух или более операторов;
- дихотомического выбора;
- повторения;

- Эдсгер Дейкстра [Дейкстра 1975] предложил отказаться от оператора безусловного перехода и ограничиться тремя конструкциями: последовательностью, выбором и циклом;
- Дональд Кнут продемонстрировал случаи, в которых оператор безусловного перехода оказывался полезным (например, выход из нескольких вложенных циклов) и подверг критике утверждение Дейкстры.

Чем еще интересны структурированные программы? В 1965 году академик Виктор Михайлович Глушков обратил внимание на то, что структурированные программы можно рассматривать как формулы в некоторой алгебре. Зная правила преобразования выражений в такой алгебре, можно осуществлять глубокие формальные (и, следовательно, автоматизированные) преобразования программ.

> ***О структурном программировании***
>
> Структурное программирование — не самоцель, его основное назначение — получение хорошей программы. Однако даже в самой хорошей программе операторы перехода требуются, например, при выходе из множества вложенных циклов.

4.3.3.2. Визуальное структурное программирование

Визуальное структурное программирование также базируется на классической работе [Дейкстра 1975]. Дейкстра предлагает ограничить топологию блок-схем, запретив проведение стрелок из любого блока в любой другой блок. Идея визуального структурного программирования — перейти от одномерного линейного представления программы к двумерному, графическому. Языки такого программирования — это языки моделирования. На сегодняшний день наиболее известны три из них.

- *Язык блок-схем ограниченной топологии.* То есть большое многообразие блок-схем должно быть приведено к трем типам операторов управления.
- *Язык схем Вельбицкого* [Вельбицкий 1984]. Эти схемы отличаются от большинства национальных стандартов построения блок-схем тем, что используется граф, нагруженный по дугам, а традиционная технология использует граф, нагруженный по вершинам (узлам). На рис. 4.9, *а* и *б* представлены два примера, показывающих представление условного и циклического предложения в виде схем Вельбицкого, соответственно. Такие схемы примерно на 25% компактнее классических блок-схем.

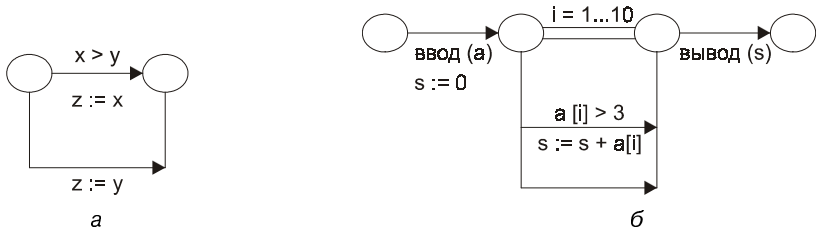


Рис. 4.9. Схемы Вельбицкого: условное (*а*) и циклическое (*б*) предложение

- *Язык схем ДРАКОН* [Паронджанов 1999]. В этом языке используются два типа элементов: графические фигуры (графоэлементы) и текстовые надписи (текстоэлементы). Одновременное использование графики и текста стимулирует одновременно и логическое, и образное мышление. Фактически существует целое семейство языков ДРАКОН:
  - ДРАКОН-1 — визуальный аналог обычного текстового псевдокода;
  - ДРАКОН-2 — язык визуального программирования систем реального времени, элемент CASE-технологии;
  - гибридные визуальные языки программирования, например ДРАКОН-ПАСКАЛЬ. В таких языках к визуальному синтаксису языка ДРАКОН по определенным правилам присоединяется текстовый синтаксис (например, языка Pascal).

Графоэлементы языка ДРАКОН называются иконами, которые могут объединяться в составные иконы — макроиконы. Шампур-блок — часть ДРАКОН-схемы, имеющая один вход сверху и один выход снизу, элементы, расположенные на одной вертикали. Терминатор — часть ДРАКОН-схемы, имеющая либо один выход снизу и не имеющая входа, либо один вход сверху и не имеющая выхода. Ветки — смысловые блоки, куски алгоритма, представляющие составной оператор языка ДРАКОН. С помощью веток можно формализовать смысловое разбиение проблемы, программы или процесса на части и дать частям удобные смысловые названия. На рис. 4.10 приводится пример ДРАКОН-схемы с двумя ветками.

Обратим внимание на то, что долгое время в Америке отвергалась идея визуального программирования, и основной упор там делался на создание универсального языка программирования.

О графических нотациях

Обратим внимание на то, что графические средства хороши, в первую очередь, для представления небольших объектов. С их ростом обозримость и удобство ссылок катастрофически падают. Вспомним, например, как непросто бывает найти нужный географический объект по ссылке на квадрат карты!

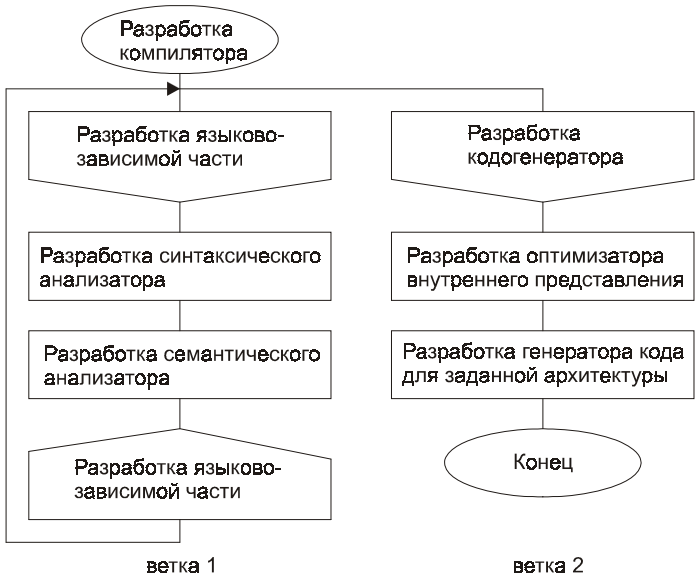


Рис. 4.10. Пример ДРАКОН-схемы с двумя ветками

4.3.3.3. Оператор ветвления

Оператор ветвления обеспечивает возможность альтернативного выполнения одного из двух или более операторов. С точки зрения языка высокого уровня выделяют:

- условный оператор (предусматривающий два варианта пути). Обычно в языках программирования он имеет синтаксис

```
if (условие) then действие1 else действие2;
```

- оператор выбора (многопутевой). Его типичный синтаксис

```
switch дискриминант case значение действие case ...;
```

Укажем некоторые особенности условного оператора в разных языках программирования:

- ряд языков (C и Algol-68) позволяют использовать присваивание в контексте выражений. Таким образом, в условии может быть присваивание, например:

```
if ( x = y < z) ...;
```

- правило языка C: часть `else` принадлежит ближайшему оператору `if`, к которому может быть присоединена;
- ряд языков программирования имеют завершающую "скобку" для условного предложения. Например, `if` — `fi` в языке Algol-68 и `if` — `end if` в языке Ada;

- в языке программирования Ada существуют два различных условных предложения: одно с частью `else`, а другое без нее.

4.3.3.4. Оператор цикла

Оператор цикла обеспечивает механизм для повторения действий. Оператор состоит из тела и заголовка, управляющего количеством повторений тела. Многие языки поддерживают операторы цикла `for`, `while ... do ...` и `repeat ... until`.

Под *итерацией* будем понимать обращение ко всем элементам агрегата в определенном порядке, причем каждый элемент посещается единожды. Самый простейший пример итератора — оператор `for`. В некоторых языках программирования итератор традиционно представляется в виде процедуры. Так, например, на языке CLU может быть написано суммирование всех элементов множества `s` с помощью итератора `elements(s)` (листинг 4.1).

Листинг 4.1. Пример использования итератора

```
setsum = proc(s:intset)returns(int)signals(NumOverFlow)
    sum: int := 0
    for i: int in intset$elements(s) do
       sum := sum + i
    end
    return(sum)
end setsum
```

Итераторы могут быть положены в основу декларативного программирования. Так можно на декларативном языке указать множество жителей города Старый Петергоф:

```
which (x: x живет в Старом Петергофе)
```

В языках программирования, предназначенных для разработки систем реального времени, отношение к итераторам особое.

- Язык не должен поддерживать циклы `while`.
- Цикл `for` разрешается использовать только с константными параметрами.
- Для периодических событий используется специальная конструкция, например `every(...) do{...}`.

4.3.3.5. Оператор исключения

Говорят, что в программе имеет место *исключение*, когда выполнение программы прерывается из-за возникновения некоторого события. Исключения, как правило, должны обрабатываться программным образом, после чего исполнение программы должно продолжаться дальше. Наиболее важно, чтобы наступление событий обрабатывалось соответствующим образом в системах реального времени (например, в программном обеспечении атомных электростанций или космических кораблей, находящихся в длительном космическом перелете).

Одно из наиболее подробных исследований принципов реализации исключений было выполнено Страуструпом [Страуструп 2000]. В частности, им обсуждались две стратегии обработки исключений в языке C++.

- *Стратегия с возобновлением*, в которой обработчик исключения может потребовать возобновления исполнения программы с того момента, где было возбуждено исключение.
- *Стратегия с прекращением исполнения*. Именно она в конечном итоге была принята, поскольку реальный практический опыт использования исключений в больших системах свидетельствовал в ее пользу.

В общем случае следует обращать внимание на реализационные особенности исключений, приведенные ниже.

- Могут ли системные исключения быть перекрыты пользователем?
- Каков механизм обработки пользовательских исключений?
- Как определяется область действия участка кода, в котором происходит событие и, следовательно, обрабатывается исключение?
- Допускаются ли в языке множественные исключения?
- Можно ли передать параметры обработчику исключения?
- Существует ли автоматическая (по умолчанию) обработка исключений?

Обратим внимание на то, что многие широко распространенные языки программирования (например, Pascal) не имеют механизма обработки прерываний.

Понятие обработки событий впервые появилось в языке PL/1. Система прерываний (так это называлось в PL/1) была предназначена для проверки и обработки различных условий, которые могут возникнуть в ходе выполнения программы. Основные категории условий в языке PL/1 следующие:

- вычислительные условия (например, переполнение или деление на ноль);
- условия ввода-вывода (например, конец файла или аппаратная ошибка устройства ввода-вывода);
- программные условия (например, выход индекса за границу массива или изменение значения некоторой переменной).

Программист может явным образом разрешать или запрещать проверку почти всех условий с помощью явных префиксов, приписываемых к инструкциям, блокам или подпрограммам. Например, в данном случае в блоке будет выполняться проверка на выход индекса за границы массива:

```
(subscriptrange): begin
    ...
    end;
```

Язык "Автокод Эльбрус" имеет конструкцию, называемую структурное предложение. Это наиболее развитый на сегодняшний день механизм обработки событий, который образно можно назвать "`goto` с человеческим лицом". Основные ситуации в языке "Автокод Эльбрус" таковы:

- *статические события*, непосредственно связанные со статической структурой текста программы (например, достижение необходимой точности вычислений);
- *динамические события*, в которых отсутствует фиксированная связь с определенной точкой в тексте программы (например, достижение конца файла для совокупности процедур работы с файлами);
- *стандартные события*, приводящие к прерываниям при выполнении стандартных операций (например, переполнение или деление на ноль). Они приводят к прерываниям процессора и обращению к операционной системе (собственно, в этом их отличие от динамических ситуаций).

Пример структурного предложения языка "Автокод Эльбрус" выглядит так:

```
до c1, c2
    % закрытое предложение
    ...
при
    c1: <обработчик 1>,
    c2: <обработчик 2>
всесит
```

Структурный переход по ситуации `c1` в закрытом предложении может быть выполнен, например, следующим образом: `c1!`.

4.3.3.6. Зависимости по управлению и по данным

Важную роль при анализе возможности распараллеливания программы играют зависимости по управлению и по данным. Для их анализа может быть построен граф программной зависимости. Его вершинами являются операторы и предикатные выражения, а дуги указывают значения данных, от которых зависят операции, и управляют условиями, от которых зависит выполнение операции.

Приведем пример *зависимости по управлению*:

```
if ( c1 )
    { s1; } /*исполнение этого оператора зависит от c1 */
s2;
```

Существуют три разновидности *зависимости по данным*.

- *Истинная (потоковая) зависимость по данным*. Значение переменной, вычисляемое в операторе `s1`, используется в операторе `s2`. Операторы `s1` и `s2` нельзя выполнять параллельно.

```
S1:  a := 3;
S2:  b := a;
S3:  c := a + 3;
```

- *Антизависимость*. Переменная в операторе `s1` используется до того, как получает новое значение в `s2`. Операторы `s1` и `s2` нельзя выполнять параллельно.

```
S1:  a := b + 3;
S2:  b := 47;
```

- *Выходная зависимость*. Переменная в операторе `s3` получает новое значение после того, как уже получила некоторое значение в `s1`.

```
S1:  a := x + 2;
S2:  b := a - 3;
S3:  a := c - 45;
```

Множество всех зависимостей задает на операторах и предикатах частичный порядок, которому нужно следовать, чтобы сохранить семантическую целостность программы.

4.3.4. Абстракция модульности

4.3.4.1. Модульное программирование

Модульное программирование — это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой. Единственная альтернатива модульности — монолитная программа, что, конечно, неудобно. Таким образом, наиболее интересным вопросом при изучении модульности является определение критерия разбиения на модули. В основе модульного программирования лежат три основные концепции.

- *Принцип утаивания информации* Парнаса. Всякий компонент утаивает единственное проектное решение, т. е. модуль служит для утаивания

информации. Подход к разработке программ заключается в том, что сначала формируется список проектных решений, которые особенно трудно принять или которые, скорее всего, будут меняться. Затем определяются отдельные модули, каждый из которых реализует одно из указанных решений.

- *Аксиома модульности* Коуэна. Модуль — это независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы. Программная единица должна удовлетворять следующим условиям:
  - *блочность организации*, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;
  - *синтаксическая обособленность*, т. е. выделение модуля в тексте синтаксическими элементами;
  - *семантическая независимость*, т. е. независимость от места, где программная единица вызвана;
  - *общность данных*, т. е. наличие собственных данных, сохраняющихся при каждом обращении;
  - *полнота определения*, т. е. самостоятельность программной единицы.
- *Сборочное программирование* Цейтина. Модули — это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:
  - стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея должна быть оформлена в виде модуля;
  - потребность организационного расчленения крупных разработок;
  - возможность параллельного исполнения модулей (в контексте параллельного программирования).

4.3.4.2. Определения модуля и его примеры

Приведем несколько дополнительных определений модуля.

- *Модуль* — это совокупность команд, к которым можно обратиться по имени.
- *Модуль* — это совокупность операторов программы, имеющая граничные элементы и идентификатор (возможно агрегатный).

Функциональная спецификация модуля должна включать:

- синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;
- описание семантики функций, выполняемых модулем по каждому из его входов.

Существуют три основные разновидности модулей.

- *Функциональные ("маленькие") модули*, реализующие, как правило, одну какую-либо определенную функцию. Основным и простейшим модулем практически во всех языках программирования является процедура или функция.
- *Информационные ("средние") модули*, реализующие, как правило, несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Примеры "средних" модулей в языках программирования следующие:
  - задачи в языке программирования Ada;
  - кластер в языке программирования CLU;
  - классы в языках программирования C++ и Java.
- *Логические ("большие") модули*, объединяющие набор "средних" или "маленьких" модулей. Примеры "больших" модулей в языках программирования следующие:
  - модуль в языке программирования Modula-2;
  - пакеты в языках программирования Ada и Java.

4.3.4.3. Характеристики модульности

Первый набор характеристик предложен Ричардом Хольтом (Richard Holt) [Holt 1975]. Это два общих требования.

- Хороший модуль снаружи лучше, чем внутри.
- Хороший модуль легче использовать, чем построить.

Второй набор предложен Гленфордом Майерсом (Glenford Myers) [Майерс 1980]. Он состоит из следующих конструктивных характеристик:

- размера модуля;
- прочности (связности) модуля;
- сцепления модуля с другими модулями;
- рутинности (идемпотентности, независимости от предыдущих обращений) модуля.

Набор этих характеристик мы далее рассмотрим более подробно.

Размер модуля

Приведем несколько общих советов по наилучшему размеру модуля.

- В модуле должно быть 7 (+/–2) конструкций (например, операторов для функций или функций для пакета).

О необходимости именно 7 (+/–2) конструкций

Это число берется на основе представлений психологов о среднем оперативном буфере памяти человека. Символьные образы в человеческом мозгу объединяются в "чанки" — наборы фактов и связей между ними, запоминаемые и извлекаемые как единое целое. В каждый момент времени человек может обрабатывать не более 7 чанков.

- Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора.

Связность (прочность) модуля

Существует гипотеза о глобальных данных, утверждающая, что глобальные данные вредны и опасны. Идея глобальных данных дискредитирует себя так же, как и идея оператора безусловного перехода `goto`. Локальность данных дает возможность легко читать и понимать модули, а также легко удалять их из программы.

Связность (прочность) модуля (cohesion) — мера независимости его частей. Чем выше связность модуля, тем лучше, тем больше связей по отношению к оставшейся части программы он упрятывает в себе. Можно выделить следующие типы связности.

- *Функциональная* связность. Модуль с функциональной связностью реализует одну какую-либо определенную функцию и не может быть разбит на 2 модуля с теми же типами связностей.
- *Последовательная* связность. Модуль с такой связностью может быть разбит на последовательные части, выполняющие независимые функции, но совместно реализующие единственную функцию. Например, один и тот же модуль может быть использован сначала для оценки, а затем для обработки данных.
- *Информационная* (коммуникативная) связность. Модуль с информационной связностью — это модуль, который выполняет несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Эта информационная связность применяется для реализации абстрактных типов данных.

О средствах задания информационной связности

Обратим внимание на то, что средства для задания информационно прочных модулей отсутствовали в ранних языках программирования (например, FORTRAN и даже в оригинальной версии языка Pascal). И только позже, в языке программирования Ada, появился пакет — средство задания информационно прочного модуля.

При написании программ следует применять по возможности лишь три вышеперечисленных типа связности. Приведенных ниже типов связности следует стараться избегать.

- *Процедурная* связность получается, когда действия сгруппированы вместе благодаря тому, что они выполняются в течение одной и той же части цикла или процесса.
- *Временная* связность представляет действия, связанные во времени. Например, в модуле объединяются действия, которые должны быть выполнены в определенный момент времени.
- *Логическая* связность возникает, когда в модуль объединяются действия только по признаку их некоторого подобия. Например, в один модуль объединяются функции, предназначенные для проверки корректности входных данных для всей программы. Как исключение, логическая связность естественна и допустима только для "больших" модулей.
- *Случайная* связность возникает тогда, когда действия объединяются произвольным, случайным образом.

Сцепление модулей

Сцепление (coupling) — это мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

- *Независимые* модули — это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого. Например, в операционной системе Unix это можно сделать с помощью каналов (pipes) для ничего не знающих друг о друге модулей `a` и `b` следующим образом: `a | b`. Достичь такого сцепления сложно, да и не нужно, поскольку сцепление по данным (параметрическое сцепление) является достаточно хорошим.
- Сцепление *по данным* (параметрическое) — это сцепление, когда данные передаются модулю, как значения его параметров либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам). Две разновидности этого сцепления определяются характером данным.
  - Сцепление по простым элементам данных.
  - Сцепление по структуре данных. В этом случае оба модуля должны знать о внутренней структуре данных.

Оставшиеся три типа сцепления использовать не рекомендуется.

- Сцепление *по управлению* ("флажковая зависимость") — это сцепление, в котором один модуль управляет решениями внутри другого с помощью передачи флагов, переключателей или специальных кодов. Таким образом,

вызывающий модуль должен достаточно подробно знать структуру вызываемого.

- Сцепление *по внешним ссылкам* возникает, когда у одного модуля есть доступ к данным в другом модуле.
- Сцепление *по кодам* возникает, если коды инструкций модулей перемежаются друг с другом. Обычно эта ситуация возникает, когда одному из модулей доступна внутренняя область другого. Сцепление по кодам позволено в языке программирования PL/1.

Рутинность модуля

Рутинность — это независимость модуля от предыдущих обращений к нему (от предысторий). Будем называть модуль рутинным, если результат его работы зависит только от количества переданных параметров (а не от количества обращений).

Модуль должен быть рутинным в большинстве случаев, но возникают случаи, когда модуль должен сохранять историю. В выборе степени рутинности модуля пользуются тремя рекомендациями.

- В большинстве случаев нужно делать модуль рутинным, т. е. независимым от предыдущих обращений.
- Зависящие от предыстории модули следует использовать только в тех случаях, когда это необходимо для сцепления по данным.
- В спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы пользователи имели возможность прогнозировать поведение такого модуля.

4.4. Языки моделирования

4.4.1. Моделирование на основе структурной методологии

В этом разделе приведены графические нотации некоторых информационных моделей, поддерживающих методы структурной методологии. Описание и предназначение методов уже было дано в *разд. 3.2.3*. Поскольку нотация языков моделирования значительно проще нотации языков программирования, то можно достаточно компактно привести изображения базовых строительных блоков (без правил, определяющих, как эти блоки могут сочетаться между собой). Мы предлагаем использовать материал данного раздела в качестве краткого справочника.

4.4.1.1. Диаграммы потоков данных

На рис. 4.11 приведена нотация диаграммы потока данных в варианте Гейна-Сарсона.

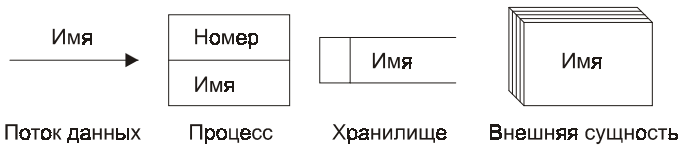


Рис. 4.11. Нотация диаграммы потока данных (вариант Гейна-Сарсона)

На рис. 4.12 также приведена нотация диаграммы потока данных, но в варианте Йордона.

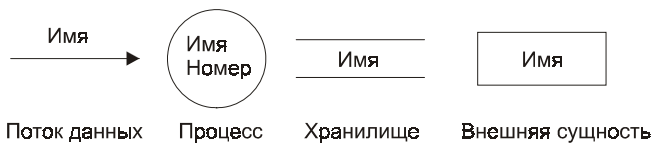


Рис. 4.12. Нотация диаграммы потока данных (вариант Йордона)

4.4.1.2. Диаграммы функционального моделирования

На рис. 4.13 приведена нотация диаграммы функционального моделирования.

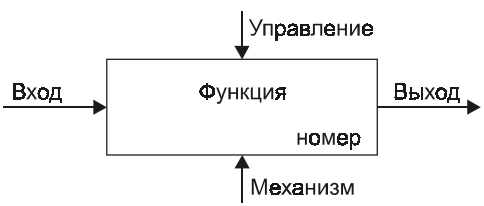


Рис. 4.13. Нотация диаграммы функционального моделирования

На рис. 4.14 изображено пять типов взаимосвязей между блоками.

- Отношение Управления (рис. 4.14, *а*).
- Отношение Входа (рис. 4.14, *б*).
- Отношение Управленческой Обратной Связи (рис. 4.14, *в*).
- Отношение Входной Обратной Связи (рис. 4.14, *г*).
- Отношение Выход-Исполнитель (рис. 4.14, *д*).

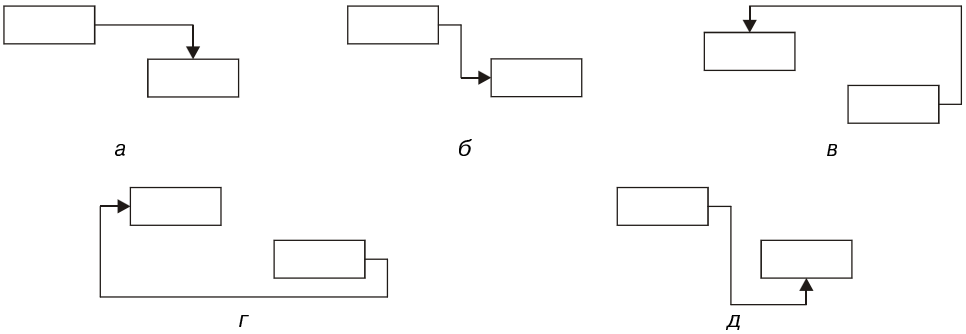


Рис. 4.14. Пять типов взаимосвязей между блоками

4.4.1.3. Диаграммы "сущность-связь"

На рис. 4.15 приведена нотация диаграммы "сущность-связь".

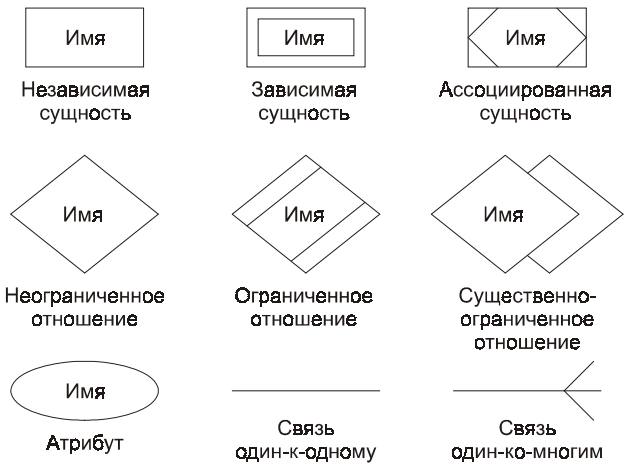


Рис. 4.15. Нотация диаграммы "сущность-связь"

4.4.2. Моделирование на основе объектно-ориентированной методологии

В этом разделе мы приведем графические или текстовые нотации некоторых информационных моделей, поддерживающих методы объектно-ориентированной методологии. Описание и предназначение методов также было дано в *разд. 3.2.3*. Еще раз подчеркнем, что поскольку нотация языков моделирования значительно проще нотации языков программирования, то можно достаточно компактно привести изображения базовых

строительных блоков (без правил, определяющих, как эти блоки могут сочетаться между собой). Предлагается использовать материал данного раздела в качестве краткого справочника.

4.4.2.1. Язык описания архитектур ACME

Данный язык имеет и текстовую, и графическую нотацию представления элементов.

Тестовое формализованное описание системы клиент-сервер на языке ACME [Ахтырченко, Леонтьев 2001] приведено в листинге 4.2.

Листинг 4.2. Пример описания системы на языке ACME

```
System simple_cs={
    Component client={Port send-request}
    Component server={Port receive-request}
    Connector rpc={Roles{caller, receiver}}
    Attachments:{
       Client.send-request to rpc-caller;
       Server.receive-request to rpc-receiver}
}
```

4.4.2.2. КОК-карты (класс-ответственность-кооперация)

На рис. 4.16 приведена нотация карт класс-ответственность-кооперация.

Имя класса	
Ответственность 1	Кооперация 1
Ответственность 2	Кооперация 2
Ответственность 3	Кооперация 3

Рис. 4.16. Нотация КОК-карт

4.4.2.3. Язык UML

Язык UML включает три вида строительных блоков [Буч, Рамбо, Джекобсон 2000].

- *Сущности* — абстракции, являющиеся основными элементами модели. В языке есть четыре типа сущностей:
  - структурные сущности (класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел);
  - поведенческие сущности (взаимодействие и автомат);

- группирующие сущности (пакет);
- аннотационные сущности (примечания).

❒ *Отношения* — основные связующие строительные блоки. Язык вводит четыре базисных отношения:

- зависимость;
- ассоциацию;
- обобщение;
- реализацию.

❒ *Диаграммы* — графические представления набора элементов, изображаемых чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями). Вообще говоря, диаграммы могут содержать любые комбинации сущностей и отношений. Однако на практике обычно применяется сравнительно небольшое количество типовых комбинаций, которые мы рассмотрим ниже.

4.4.2.4. Диаграммы классов

На рис. 4.17, *а—в* приведена нотация диаграммы классов.

Для отдельных классов сначала может быть указано только имя класса (рис. 4.17, *а*), а при дальнейшей разработке добавлены атрибуты и операции класса (рис. 4.17, *б*). На рис. 4.17, *в* изображена схема задания отношения бинарной ассоциации между классами.

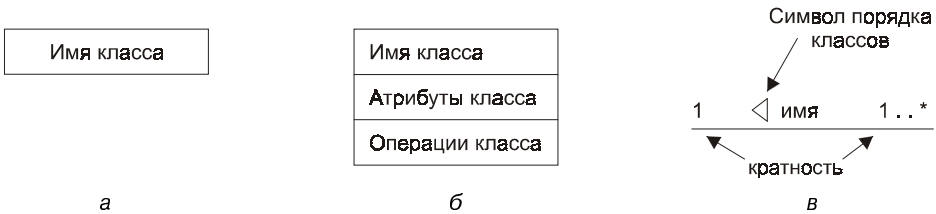


Рис. 4.17. Нотация диаграммы классов

Нотация четырех типов отношений между классами приведена на рис. 4.18. Каждому типу отношений соответствует свое изображение связи.

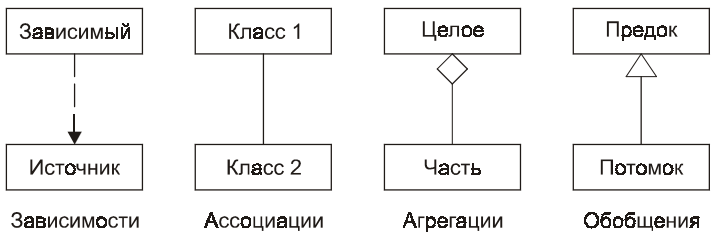


Рис. 4.18. Четыре типа отношений между классами

4.4.2.5. Диаграммы вариантов использования

На рис. 4.19 приведена нотация диаграммы вариантов использования.

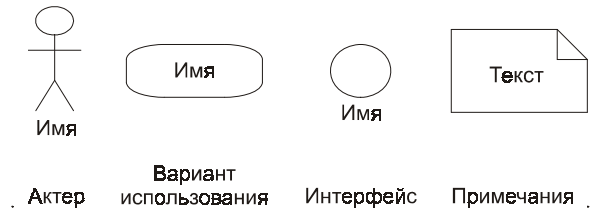


Рис. 4.19. Нотация диаграммы вариантов использования

4.4.2.6. Диаграммы состояний

На рис. 4.20, *а*—*г* приведена нотация диаграммы состояний.

Для отдельных состояний может быть указано либо только имя состояния (рис. 4.20, *а*), либо имя и список внутренних действий или переходов в данном состоянии (рис. 4.20, *б*). На рис. 4.20, *в* приведен переход между состояниями, а на рис. 4.20, *г* — графическое изображение начального и конечного состояний.

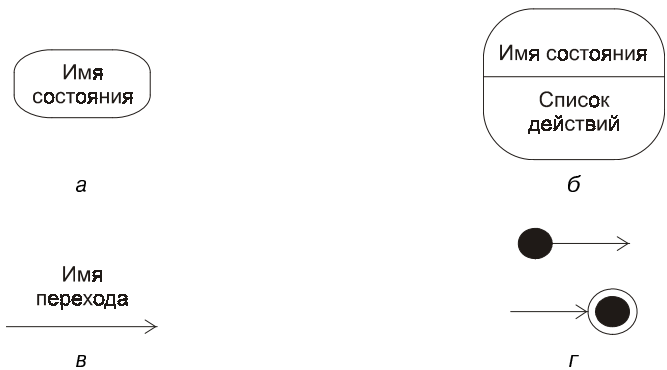


Рис. 4.20. Нотация диаграммы состояний

4.4.2.7. Диаграммы деятельности

На рис. 4.21, *а*—*г* приведена нотация диаграммы деятельности.

На рис. 4.21, *а* приведено графическое изображение состояния действия. На рис. 4.21, *б* приведен пример ветвления, а на рис. 4.21, *в* и *г* — графическое изображение разделения и слияния параллельных потоков управления.

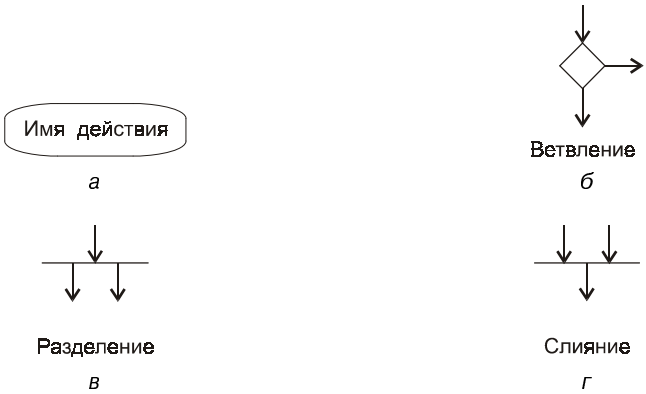


Рис. 4.21. Нотация диаграммы деятельности

4.4.2.8. Диаграммы последовательности

На рис. 4.22, *а* приведен простейший графический примитив диаграммы последовательности. Фокус управления изображается в форме вытянутого узкого прямоугольника (рис. 4.22, *б*).

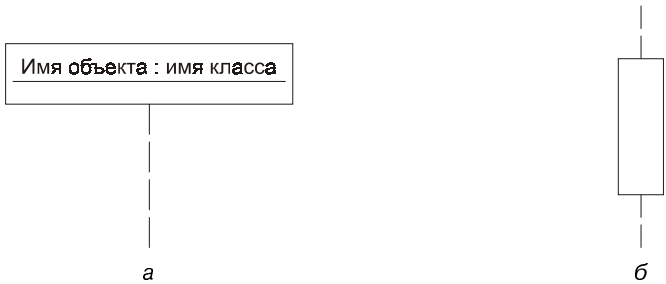


Рис. 4.22. Нотация диаграммы последовательностей

На рис. 4.23 приведено графическое изображение различных видов сообщений между объектами на диаграмме.

Рис. 4.23. Виды сообщений между объектами на диаграмме последовательностей

4.4.2.9. Диаграммы кооперации

На рис. 4.24 и 4.25 приведена нотация диаграмм кооперации уровня спецификации и уровня примеров соответственно.

Рис. 4.24. Нотация диаграммы кооперации уровня спецификации

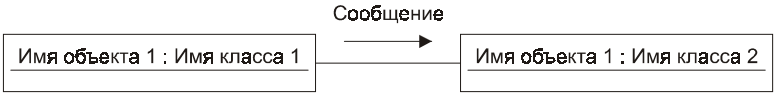


Рис. 4.25. Нотация диаграммы кооперации уровня примеров

4.4.2.10. Диаграммы компонентов

На рис. 4.26 приведена нотация диаграммы компонентов.

Рис. 4.26. Варианты графического изображения компонентов

Компоненты различных видов имеют свои изображения:

- обычное изображение компонента (рис. 4.27, *а*);
- динамически подключаемые библиотеки (рис. 4.27, *б*);
- страницы на языке разметки гипертекста (рис. 4.27, *в*);
- файлы справки (рис. 4.27, *г*);
- файлы с исходными текстами программ (рис. 4.27, *д*).

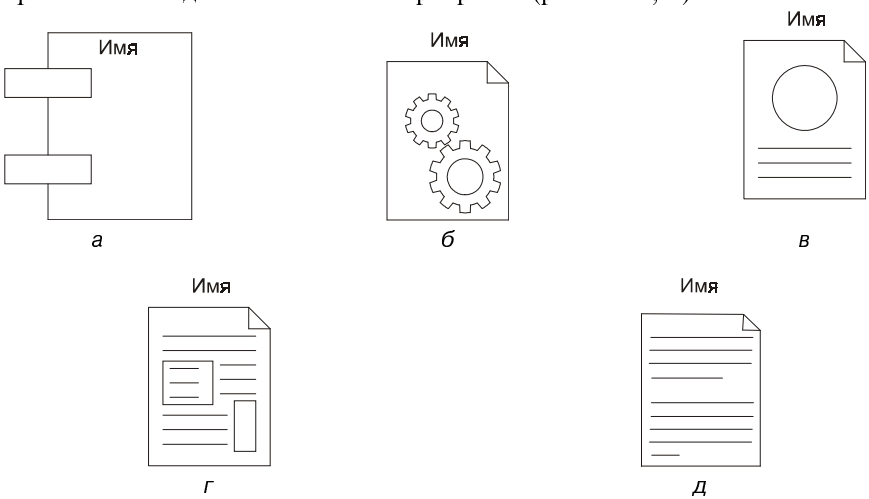


Рис. 4.27. Нотация диаграммы компонентов

4.4.2.11. Диаграммы развертывания

На рис. 4.28 приведен пример использования нотации диаграммы развертывания.

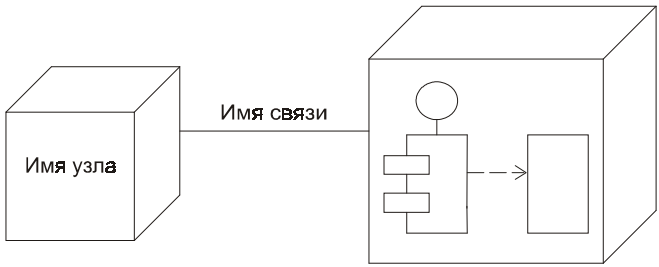


Рис. 4.28. Пример использования нотации диаграммы развертывания

4.4.3. Языки моделирования данных

4.4.3.1. Модель данных

Мы уже отметили, что данные — это отдельные элементы, которые могут быть собраны вместе некоторым образом. Термин "данные" происходит от латинского "datum", означающего "факт".

Моделью данных будем называть средство, позволяющее реализовать интерпретацию данных в соответствии с указанными требованиями. Модель данных является средством абстрагирования, дающим возможность увидеть информационное содержание данных, а не их конкретные значения. Модели данных разделяются на два класса.

- *Сильно типизированные* модели, в которых предполагается, что все данные должны быть отнесены к какой-либо категории.
- *Слабо типизированные* модели, не связанные никакими предположениями относительно категорий.

Модель данных определяет правила порождения допустимых структур данных и возможные операции над такими структурами. Это связано с ограничениями, вытекающими из особенностей используемых типов структур данных и операций, которые можно выполнять над структурами. Структуризация основывается на использовании отношений типа обобщение и агрегация.

О модели данных "сущность-связь"

Инструментальные средства для спецификации концептуальной модели предметной области также принято называть моделями данных. Наиболее известной из таких моделей является модель "сущность-связь".

Языки моделирования данных являются комбинацией, по крайней мере, двух языков:

- *языка определения данных*, который поддерживает определения или объявления данных;
- *языка обработки данных*, который поддерживает операции с такими объектами или их обработку. Эти языки по способу получения результата подразделяются на два класса:
  - навигационные языки, которые осуществляют последовательное прохождение по связям, реализованным в структуре модели данных;
  - спецификационные языки, которые определяют только требования к результату, но не задают способ его получения.

Основные компоненты модели данных следующие:

- структуры данных;
- операции над данными;
- ограничения целостности (логические операции, накладываемые на данные).

4.4.3.2. Реляционная модель данных

Реляционная модель данных — это способ рассмотрения данных, т. е. предписание для способа представления данных (посредством таблиц) и для способа работы с таким представлением (посредством операторов) [Дейт 2000].

Реляционную модель можно рассматривать как набор двумерных таблиц, где вертикальные колонки соответствуют элементам данных или полям, а горизонтальные строки объединяют поля в единые объекты. Разработка модели включает идентификацию объектов, их атрибутов и первичных ключей.

Первичный ключ — это атрибут или набор атрибутов, который может быть использован для однозначной идентификации строки таблицы.

Основными проблемами работы с данными является достижение непротиворечивости данных и отсутствие их дублирования. Эти задачи решаются с помощью нормализации. Концепции и методы нормализации были разработаны Эдгаром Коддом (Edgar Codd). На рис. 4.29 изображен процесс нормализации.

Процесс нормализации, вообще говоря, не ограничивается третьей нормальной формой, но ее, в большинстве случаев, вполне достаточно для обеспечения непротиворечивости и отсутствия дублирования.

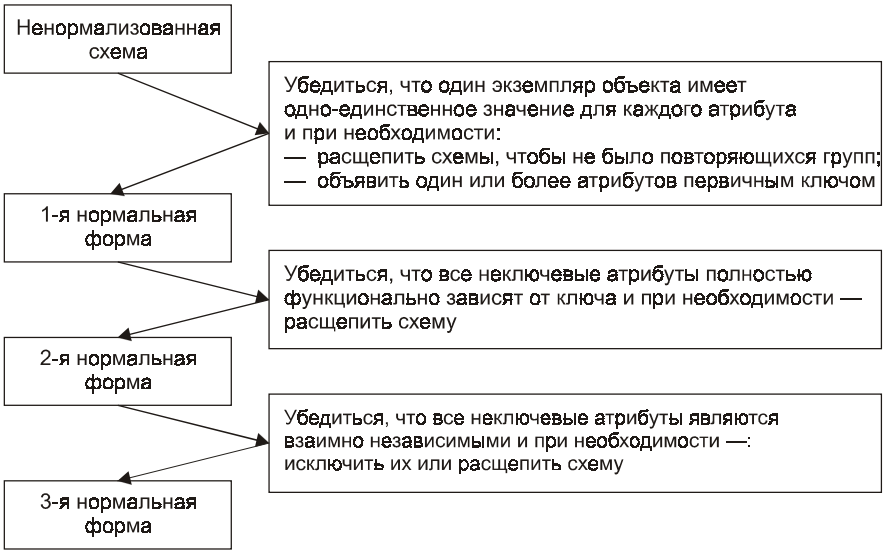


Рис. 4.29. Процесс нормализации

4.4.3.3. Другие модели данных

В первой половине 70-х годов XX века велись споры о преимуществах трех моделей данных.

- Реляционной.
- Сетевой.
- Иерархической.

Эти споры завершились "ничейным" исходом в открытой дискуссии Бахмана и Кодда в 1975 году. В настоящее время сформировалась новая тройка конкурентов.

- Реляционная модель.
- Объектная модель.
- Многомерная модель.

4.4.4. Языки моделирования знаний

4.4.4.1. Классики — о знаниях

Знание — это препятствие на пути неизвестности.
С. Паронен

Знание — особый, специфический вид информации. Знание характеризуется очень важным свойством — *активностью*, т. е. способностью дать его

носителю обрабатывать информацию. Знание традиционно разделяется на два вида: декларативное и процедурное.

- *Декларативное* знание — это набор конкретных сведений, относящихся к некоторой ситуации и фигурирующим в ней объектам.
- *Процедурное* знание — набор предписаний на какие-то случаи жизни. Это могут быть методы, алгоритмы и программы решения различных задач, с которыми человек уже сталкивался раньше и научился их решать.

Существует ряд более детальных классификаций. Например, классификация Лаврова [Лавров 1986] включает следующие типы знаний:

- *конструктивное* знание — знание о возможной структуре и взаимодействии частей различных объектов;
- *фактографическое* знание — количественные и качественные характеристики конкретных объектов, явлений и их элементов;
- *понятийное* знание — набор некоторых понятий, описание их свойств и взаимосвязей. Это "первичное" знание, можно сказать, что это знание об основах знаний, знание о том, как использовать знания.

4.4.4.2. Модели представления знаний

Существует две основных модели представления знаний: логическая и эвристическая [Попов 1987].

В основе *логических моделей* лежит понятие формальной системы (теории). В формальных системах можно выделить множество базовых элементов, синтаксических правил, аксиом и правил вывода. Применяя правила вывода к аксиомам, можно получать синтаксически правильные совокупности, к которым можно снова применить правила. Правила вывода — наиболее сложная составляющая системы. Примеры формальных теорий — исчисление предикатов, конкретные системы продукций. В логических моделях отношения между отдельными единицами знаний выражаются только с помощью тех небогатых средств, которые предоставляются синтаксическими правилами используемой формальной системы.

В отличие от формальной модели, *эвристические модели* имеют разнообразный набор средств, передающих специфические особенности той или иной проблемной области. Здесь повышается эффективность использования правил вывода. Среди эвристических моделей можно выделить три основные группы.

- *Продукционные* модели, образованные правилами.
- *Сетевые* модели, в основе которых лежит понятие сети, образованной помеченными вершинами и дугами. Вершины представляют собой некоторые сущности, а дуги — отношения между связываемыми ими сущностями.

- *Фреймовые* модели, базирующиеся на специальных структурах — фреймах, предназначенных для представления некоторых стандартных ситуаций.

4.4.4.3. Метрики левой и правой моделей мира

Головной мозг — это орган, которым мы думаем,
будто мы думаем.
А. Бирс

Модели представления знаний во многом определяются человеческим восприятием модели мира. Как известно, мозг человека состоит из двух полушарий, каждое из которых по-своему преобразует информацию [Грановская, Березная 1991]. Данная особенность организации мозга с возрастом и развитием человека усиливается и оказывается столь существенной, что постепенно оба полушария начинают совсем по-разному участвовать во всех психических процессах.

Каждое полушарие мозга получает информацию только от одной половины поля зрения, однако образ строится целостный — обо всем поле сразу. Получая половину картины, каждое полушарие стремится к завершению образа, т. е. достраивает объект, но делает это на свой манер. Правое полушарие, сохраняя целостность образа, усиливает в нем стереотипность за счет упрощения деталей, левое удерживает отдельные детали, не сохраняя целостности.

Модель мира как представление о себе и мире составляет центральную часть психической жизни человека. Эту модель можно представить в виде взаимодействия нескольких крупных блоков: высших психических функций, структуры ценностей, системы психологической защиты. Построение внутренней модели мира осуществляется в памяти в большей мере по законам доминирующего полушария.

Память — это процесс запечатления, сохранения и воспроизведения следов прошлого опыта. Допуская экстраполяцию и интерполяцию, память раздвигает границы восприятия временных и пространственных характеристик.

Левополушарная память обнаруживает себя как хранилище знаний, выраженных словами, символами, значениями и отношениями между ними в формулах и алгоритмах. Относящаяся к ней семантическая память — это "умственный лексикон" абстрактного знания, хранимый без ссылки на обстоятельства, при которых оно приобретено. Осуществляемое в ней обобщение, с одной стороны, обогащает память за счет введения запоминаемого в рамки определенных категорий, а с другой — обедняет ее за счет сохранения в понятиях только типовых деталей.

Правополушарная память — эпизодическая, данная в контексте, в противоположность левополушарной памяти, классифицированной по различным основаниям и данной вне контекста. Использование левополушарной дает

возможность быстро узнавать, а правополушарной — произвольно воспроизводить и экстраполировать свойства объектов, повышая предсказуемость ситуации.

Все сказанное о памяти подводит нас к выводу, что специфика правого и левого полушарий определяется используемым способом объединения материала. Правополушарная классификация — ситуативная, она опирается на практический опыт человека, а левополушарная — категориальная, основанная на логике и понятийном мышлении.

4.5. Языки программирования высокого уровня

Язык программирования высокого уровня достаточно полно описывается совокупностью следующих особенностей:

- язык *обладает характеристиками*. Примеры таких характеристик приведены в *разд. 4.1.1.3*;
- язык *содержит абстракции*. Основные языковые абстракции рассмотрены в *разд. 4.3*;
- язык *поддерживает методологию* программирования. Обзор языков, поддерживающих методологии, приведен в *гл. 2*;
- язык *принадлежит к семейству* языков. Семейства подробно рассматриваются в *разд. 4.5.1*;
- язык *может быть специализированным* для некоторой предметной области. Специализированным языкам посвящен *разд. 4.5.2*.

4.5.1. Обзор языков, принадлежащих к семействам

Родственные связи между языками мы можем проследить, определяя их принадлежность к некоторому семейству. Во многих случаях можно указать родословное дерево языков, четко определяя последовательность языков, "генетически" оказавших влияние друг на друга.

4.5.1.1. Семейство универсальных языков

Большинство языков этого семейства было создано в 60-е годы XX века (рис. 4.30).

Они характеризуются:

- ориентацией на большой класс задач, в первую очередь, вычислительных;
- множеством языковых конструкций и некоторой громоздкостью;

- долгожительством. Все языки применяются в настоящее время;
- все эти языки поддерживают императивную методологию.

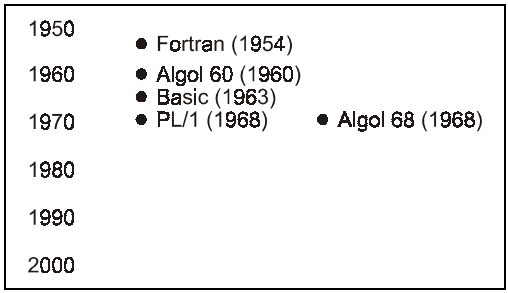


Рис. 4.30. Семейство универсальных языков

Обратим внимание на то, что такие языки, как Pascal, Ada и C, несомненно, строились как продолжатели традиций семейства универсальных языков.

4.5.1.2. Семейство уникальных языков

Уникальные языки характеризуются некоторой определенной чертой, "изюминкой", выделяющей их среди других языков. Практически все из них (рис. 4.31) остались в определенном смысле неповторимыми.

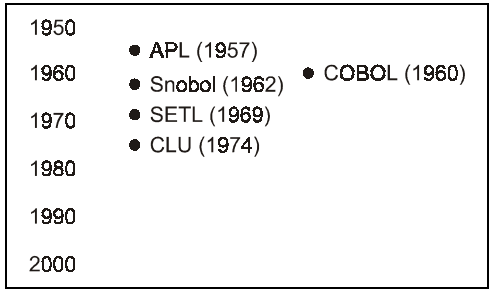


Рис. 4.31. Семейство уникальных языков

Большинство этих языков поддерживают императивную методологию. Их более поздние диалекты иногда обогащались методами поздних по возникновению методологий. Одновременно закономерным и все-таки неожиданным стало появление в конце 80-х годов XX века диалекта Object-Oriented Cobol.

Ряд языков из этого семейства можно охарактеризовать, как языки с одним типом данных.

Дадим краткую характеристику языкам этого семейства.

- APL — язык, в котором единственным структурным типом данных являются массивы: одномерные (векторы) и двумерные (матрицы).

- Cobol — язык, ориентированный на обработку коммерческой информации. На этом языке хорошо писать задачи, требующие строго фиксированного формата выдачи (печати) результатов. Можно рассматривать этот язык как непосредственного предка языка PL/1.
- Snobol — язык, предназначенный для работы со строками.
- SETL — язык, в котором одним из основных типов данных является множество.
- CLU — язык, в котором наиболее полное воплощение получила концепция абстрактных типов данных.

4.5.1.3. Семейство языков параллельного программирования

Семейство представляет собой множество параллельных языков программирования, частично обсуждавшихся в *разд. 2.4.1* и *2.4.2*. Параллельные языки различаются:

- способом организации процессов:
  - сопрограммы. Например, в языках Modula-2 и BLISS;
  - разветвления (fork) и объединения (join). Например, в языках PL/1 и Mesa;
  - параллельные скобки (cobegin/coend). Например, в языках Algol-68 и Edison;
  - объявления процессов. Например, в языках Ada и Concurrent Pascal;
- способом синхронизации процессов:
  - семафоры. Например, в языке Algol-68;
  - мониторы. Например, в языке Concurrent Pascal;
  - рандеву. Например, в языке Ada;
  - критические участки. Например, в языке Edison;
  - передача сообщений. Например, в языке CSP;
  - удаленный вызов процедур. Например, в языке DP;
  - атомарные транзакции. Например, в языке Argus.

4.5.1.4. C-семейство языков

```
for(;P("\n"),R-;P("|"))for(e-C;e-;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

Майк Тэйлор

Языки этого семейства (рис. 4.32) содержат в себе, как подмножество, языковые конструкции языка C. Дополнение языка C объектно-ориентированными конструкциями дало язык C++, а обрезание некоторых опасных, с точки зрения программирования, конструкций языка C у языка

C++ привело к появлению языка Java. Отметим, что многие концепции, присутствующие в Java (как в языке, так и в окружении), заимствованы авторами Java из проекта Oberon Никлауса Вирта.

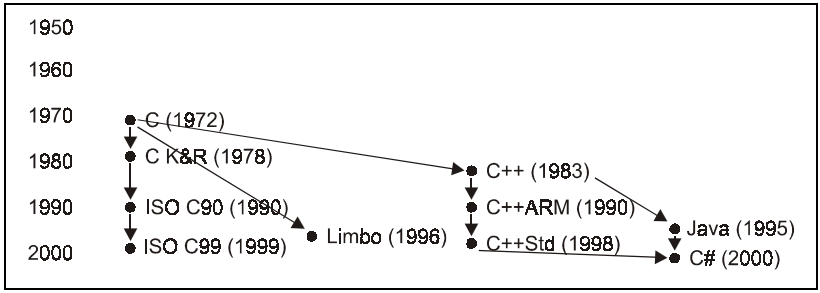


Рис. 4.32. C-семейство языков

4.5.1.5. Pascal-семейство языков

Это семейство (рис. 4.33) можно также назвать семейством Никлауса Вирта, знаменитого швейцарского профессора. Вирта часто называют "отцом структурных языков". Однако именно ему удалось создать истинно объектно-ориентированный язык программирования — Oberon, способный обеспечить безопасность типов.

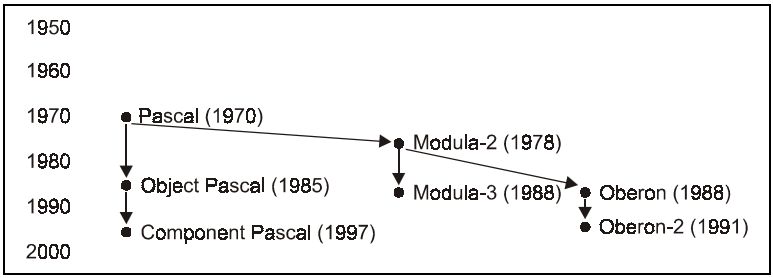


Рис. 4.33. Pascal-семейство языков

4.5.1.6. Ada-семейство языков

Ada — многофункциональный, мощный, универсальный язык. Первый его вариант создавался в период с 1979 по 1983 годы (рис. 4.34). Это один из немногочисленных языков, созданных "на заказ", причем заказчиком выступало министерство обороны США.

4.5.1.7. Simula-семейство языков

Язык Simula получил широкое применение для решения задач имитационного моделирования. Языки этого семейства (рис. 4.35) воплощают концепцию языка-ядра. Эта особенность позволяет сравнительно просто вводить

расширения языка, ориентированные на конкретные области применения. Язык Simula считается основателем, а Smalltalk — классическим представителем объектно-ориентированных языков. Язык BETA интересен тем, что он комбинирует абстракции данных, управления и процедур в одну исчерпывающую абстракцию. Объекты этого языка (порождаемые паттернами) могут рассматриваться как переменные, структуры данных, процедуры, функции, параллельные системы.

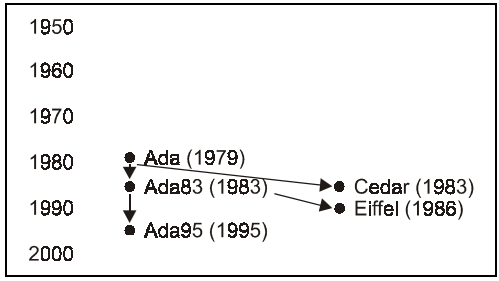


Рис. 4.34. Ada-семейство языков

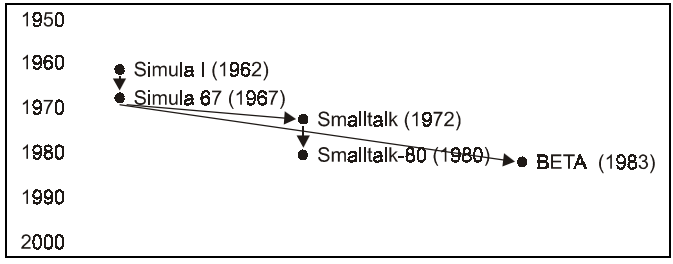


Рис. 4.35. Simula-семейство языков

4.5.1.8. Lisp-семейство языков

Это семейство (рис. 4.36) близко к языкам функционального программирования, уже рассмотренным *в разд. 2.2.3.*

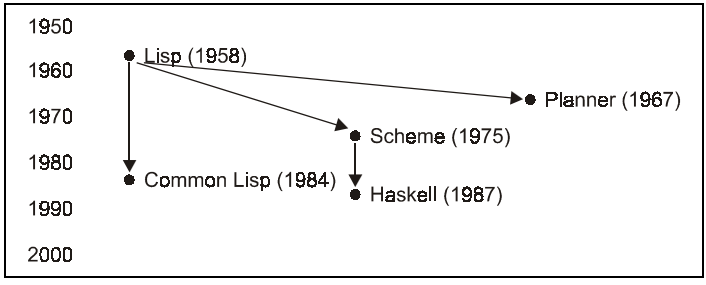


Рис. 4.36. Lisp-семейство языков

4.5.1.9. Prolog-семейство языков

Это семейство (рис. 4.37) близко к языкам логического программирования, рассмотренным ранее в *разд. 2.2.4*.

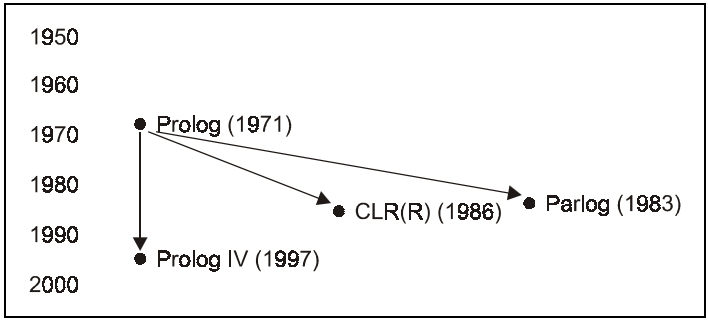


Рис. 4.37. Prolog-семейство языков

4.5.1.10. Forth-семейство языков

Это семейство базируется на языке Forth, разработанном для функционирования на компьютерах, управляющих процессами в режиме реального времени. Благодаря компактности языка, трансляторы встраивались во многие приложения с небольшими затратами и усилиями. Forth стал основой для языка PostScript. Охарактеризовать эти языки можно так:

- Forth — язык, в котором программы записываются в постфиксной записи и в стековой нотации;
- PostScript — язык, использующий простую виртуальную машину постфиксных вычислений, а также возможности раскрашивать информацию на странице.

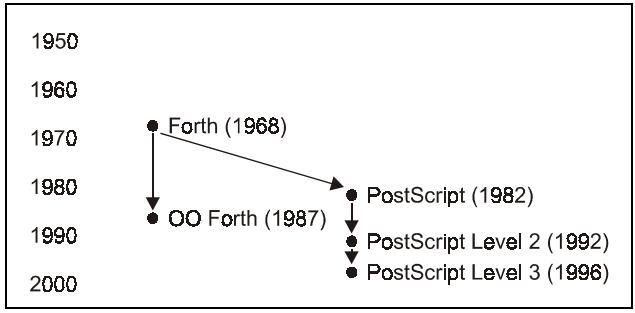


Рис. 4.38. Forth-семейство языков

4.5.2. Обзор языков, ориентированных на предметную область

Создание языка, более или менее специализированного к предметной области, — крупное достижение. Создавая такие языки, авторы должны долгое время продумывать, как наилучшим образом решать проблемы, исследовать их, смотреть на них с различных точек зрения. Предметных областей достаточно много, и не все из них имеют специализированные языки.

Программы, которые написаны на языках, ориентированных на предметную область, имеют следующие преимущества:

- они более точные;
- они могут быть быстро написаны;
- их легко сопровождать;
- о них легко говорить в понятиях и терминах предметной области;
- их могут использовать непрограммисты.

Далее мы выделим несколько областей и укажем языки, ориентированные для них.

4.5.2.1. Языки форматирования текстов

Предметная область — издательские системы, системы подготовки научных публикаций с большим количеством формул. Среди языков форматирования наиболее известны следующие:

- TeX — язык форматирования текстов, разработанный Дональдом Кнутом [Кнут 1993]. Язык использует специальные команды, которые предваряются символом начала команды для различения их и основного текста;
- LaTeX — один из языков, представляющий собой надстройку над TeX. Особенность таких надстроек в том, что они предлагают стилевые файлы, упрощающие задание и, соответственно, получение определенного результата форматирования;
- troff и nroff — форматоры текстов в операционной системе Unix [Баурн 1986].

4.5.2.2. Языки разметки

Предметная область — универсальные форматы представления структурированных данных. Язык разметки SGML (Standard Generalized Markup Language — стандартный обобщенный язык разметки) представляет собой набор правил и шаблонов для описания содержания, структуры и формата любых электронных документов, независимо от конечного способа их представления.

На основе данного языка разработан ряд специализированных языков, в том числе:

- XML (eXtensible Markup Language) — расширяемый язык разметки;
- HTML (HyperText Markup Language) — язык разметки гипертекста;
- MathML (Mathematics Markup Language) — язык описания математических формул.

4.5.2.3. Языки сценариев (скриптов)

Предметная область — разработка командных пакетных файлов для командных интерпретаторов. *Языков сценариев* (скриптов) довольно много, поэтому разделим их на три группы.

- *Командно-сценарные языки* появились в 1960-х годах, когда возникла острая потребность в языках управления заданиями. Наиболее известен язык JCL, разработанный еще для OS/360. К этой группе также относится большое количество языков создания командных пакетных файлов на основе команд операционных систем. Примерами могут служить sh, bash, csh, ksh — языки сценариев и скриптов операционной системы Unix.
- *Прикладные сценарные языки*, появившиеся в 1980-х, были ориентированы на клиентскую часть программного обеспечения. Примером является Javascript — язык сценариев текстовых браузеров.
- *Универсальные сценарные языки*, как правило, применяются в сетевой среде. Perl — язык, часто применяющийся в CGI-скриптах, а Tcl/Tk — язык скриптов графического пользовательского интерфейса.

4.5.2.4. Промежуточные языки программирования

Многие языковые трансляторы *(см. разд. 5.2.4.2)* используют промежуточные языки программирования, являющиеся императивными. *Промежуточные языки* играют важную роль в переносимости, стоимости и эффективности трансляторов. Существуют два основных способа становления языка в качестве промежуточного [Handbook 1998].

- *Включение*, когда существующий язык становится промежуточным языком некоторого транслятора. Примерами могут служить:
  - ранние компиляторы C++, которые использовали язык программирования C в качестве промежуточного языка программирования;
  - для языка разметки и верстки LaTeX промежуточным языком будет TeX.
- *Разработка*, когда промежуточный язык специально создается для использования между компонентами транслятора. Многие компании, разрабатывающие трансляторы, используют промежуточные языки для

сокращения усилий по их разработке. В таком случае вместо разработки трансляторов, например, с языков C, C++, Pascal в архитектуру Intel, PowerPC и SPARC для каждого варианта, т. е. 9 трансляторов, можно обойтись созданием лишь 6. Первые 3 программы будут выполнять анализ текстов на исходных языках и преобразовывать их в промежуточное представление. Оставшиеся 3 программы будут транслировать промежуточное представление в объектный код. Наиболее известными промежуточными языками, созданными для этих целей, являются:

- Pcode — для языка программирования Pascal;
- Java Byte Code — для языка программирования Java;
- Diana — для языка программирования Ada.

О промежуточном языке для семейства операционных систем

Идея промежуточных языков была перенесена из области разработки трансляторов в область разработки операционных систем. Для семейства операционных систем разрабатывается промежуточный объектный код. Он обеспечивает переносимость (мобильность) программ на уровне объектного кода.

4.5.2.5. Языки программирования агентов

Предметная область — многоагентные системы. *Агенты* — сущности, находящиеся в некоторой среде, от которой они получают данные, отражают события, происходящие в среде, интерпретируют их и исполняют команды, воздействующие на среду. Наиболее адекватно задачу разработки агентов решают следующие языки программирования:

- Java — универсальный язык программирования;
- AgentSpeak — язык переговоров и обмена знаниями;
- TeleScript — специализированный язык программирования агентов;
- Oz — язык описания интеллектуальных агентов.

4.5.2.6. Языки создания графики

Предметная область — подготовка визуальной графической информации. Наиболее известные языки создания графики таковы:

- уже упоминавшийся PostScript — язык управления для целого ряда графических устройств: от фотонаборных автоматов до лазерных принтеров;
- Open GL — интерфейс программирования трехмерной графики (**http://www.opengl.org.ru**).

4.5.2.7. Языки описания аппаратуры

Предметная область — описание аппаратуры микросхем. В течение многих лет широко используются следующие языки описания аппаратуры:

- VHDL (Very high speed integrated circuit Hardware Description Language) — язык описания аппаратуры на базе языка Ada. Разрабатывался для моделирования цифровых систем;
- Verilog — язык, позволяющий достаточно эффективно выполнить описание и провести моделирование и синтез цифровых схем благодаря:
  - применению встроенных примитивов;
  - применению примитивов пользователя;
  - средствам временного контроля;
  - средствам моделирования задержки распространения от входа до выхода;
  - возможности задания внешних тестовых сигналов.

4.5.2.8. Языки описания виртуальной реальности

Предметная область — создание трехмерных изображений, так называемых виртуальных миров, в которых пользователь может перемещаться и взаимодействовать с объектами. Языки позволяют определить среду действия персонажей компьютерных игр. В качестве примера приведем следующие языки описания виртуальной реальности:

- VRML (Virtual Reality Modeling Language) — язык моделирования виртуальной реальности. Язык позволяет определить реакцию объектов на события и перемещения объектов;
- Dynamo — язык описания трехмерных виртуальных миров.

4.5.2.9. Языки конфигурирования

Предметная область — совокупность управляющей информации некоторой системы и процесс адаптации системы для работы в определенных рамках, налагаемых внешним окружением и пользователем. Перечислим наиболее известные языки конфигурирования:

- Autoconf — язык макрокоманд, порождающих выполняемые командным интерпретатором скрипты, задачей которых является конфигурация исходных текстов;
- макрокоманды m4.

4.6. Формальные языки

Существуют два основных механизма определения языков [Ахо, Ульман 1978].

- Механизм порождения или генератор.
- Механизм распознавания (разбора) или распознаватель.

4.6.1. Иерархия грамматик

Грамматики образуют наиболее важный класс генераторов языков. *Грамматика* — это математическая система, определяющая язык. Наиболее известной иерархией грамматик и языков является иерархия Хомского. Эта иерархия включает четыре типа грамматик, последовательно вводящих все более сильные ограничения на форму правил грамматики [Фитиалов 1984]. К этим четырем типам добавлен еще один (в нашей нумерации это тип 4) [Grune, Jacobs 1998], достаточно широко используемый.

- Тип 0 — Грамматики *без ограничений* на вид правил (произвольные грамматики).
- Тип 1 — *Неукорачивающие* (монотонные) грамматики с правилами вида `u->v`, где `0 < |u| ≤ |v|` (неукорачивающие правила). Важный подкласс грамматик типа 1 — контекстно-зависимые грамматики с правилами вида `uAv->uwv`, где `w` — непустая цепочка.
- Тип 2 — *Контекстно-свободные* (бесконтекстные) грамматики с правилами вида `A->w`. Некоторые авторы предъявляют к контекстно-свободным грамматикам требование быть неукорачивающими. Это означает, что в таких грамматиках отсутствуют стирающие правила вида `A->e`.
- Тип 3 — *Регулярные* (автоматные, с конечным числом состояний) грамматики с правилами видов `A->e`, `A->a`, `A->aB`. То есть в таких грамматиках допустимы всего два вида правил: нетерминал, порождающий ноль или более терминалов, и нетерминал, порождающий ноль или более терминалов, за которым следует один нетерминал.
- Тип 4 — Грамматики *конечного выбора*, в которых не допускаются нетерминалы в правой части правил.

В табл. 4.2 представлены две иерархии грамматик: Хомского (монотонная) и укорачивающая (немонотонная) [Grune, Jacobs 1998].

Таблица 4.2. *Иерархия грамматик*

Типы грамматик	Иерархия Хомского (монотонная)	Немонотонная (укорачивающая) иерархия
Тип 0	Грамматики без ограничений на структуру (Монотонные грамматики со стирающими правилами)	Грамматики без ограничений на структуру
Тип 1	Контекстно-зависимые грамматики (Монотонные грамматики без стирающих правил)	Контекстно-зависимые грамматики с немонотонными правилами

Таблица 4.2 *(окончание)*

Типы грамматик	Иерархия Хомского (монотонная)	Немонотонная (укорачивающая) иерархия
Тип 2	Контекстно-свободные грамматики без стирающих правил	Контекстно-свободные грамматики
Тип 3	Регулярные грамматики (без стирающих правил)	Регулярные грамматики, регулярные выражения
Тип 4	Конечный выбор	

4.6.2. Техники распознавания (разбора)

Техника разбора позволяет связать грамматику и предложения языка. Существуют две основные техники разбора, каждая из которых воплощена во многих методах.

- Разбор *сверху вниз*, в котором дерево разбора строится от вершины (стартового символа).
- Разбор *снизу вверх*, в котором путь разбора проходится от предложения до стартового символа.

Некоторые основные техники распознавания приведены в табл. 4.3 [Grune, Jacobs 1998].

Таблица 4.3. *Техники распознавания*

Методы	Сверху вниз	Снизу вверх
Непрямые методы	Распознаватель Унгера	Распознаватель КЯК (Кока, Янгера, Касами)
Прямые методы	predict/match	shift/reduce
Линейно-направленные методы: сначала в ширину с шириной, ограниченной 1	Единственный метод: - LL(k)	Большой набор методов: - Предшествования - Контекстно-связанные - LR(k) - LALR(1) - SLR(1) - ...
Эффективные методы общей направленности: максимально ограниченный поиск сначала в ширину	(нет известных методов)	Распознаватель Томита

4.6.3. Образное сравнение типов грамматик

На рис. 4.39 представлено образное сравнение типов грамматик [Grune, Jacobs 1998]. Пусть роза на рисунке представляет язык, молекулы розы соответствуют предложениям языка. Грамматики будут обрисовывать контур розы.

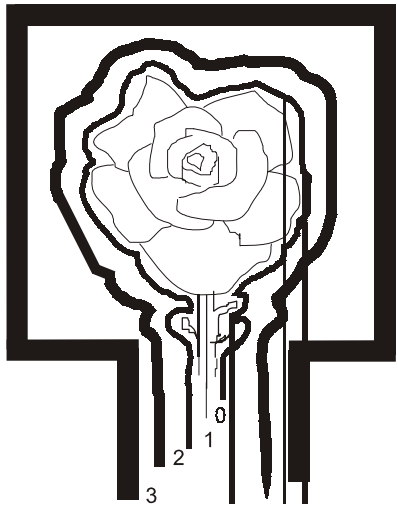


Рис. 4.39. Образное сравнение типов грамматик

- Регулярные грамматики способны описать цветок лишь с помощью грубых вертикальных и горизонтальных линий (контур 3 на рис. 4.39).
- Контекстно-свободные грамматики описывают контур немного лучше, с помощью совокупности дуг (контур 2 на рис. 4.39).
- Контекстно-зависимые грамматики представляют описание в виде более гибкой линии, которая, тем не менее, не отражает сложные участки (контур 1 на рис. 4.39).
- Грамматики без ограничений на структуру могут представить розу в виде практически идеального контура (контур 0 на рис. 4.39).

4.6.4. Метатрансляция

На сегодняшний день существует довольно много оригинальных методов осуществления трансляций, но так и не возникло никаких серьезных подходов к созданию теории метатрансляции. Если проводить параллели с математикой, то можно сказать, что информатика находится на этапе зарождения своей собственной логики или формализованной системы, сводящей воедино все богатство накопленного "сырого материала".

Во всех существующих подходах к проблеме трансляции присутствует один серьезный недостаток — трансляция всегда жестка. В каждой формализованной схеме трансляции у нас есть одна или несколько заранее зафиксированных систем, при помощи которых мы можем осуществлять трансляцию некоего входа. Как видно, переменной величиной здесь является лишь вход, и, возможно с некоторой натяжкой, проблемная область, приведшая к фиксированию именно таких, а не иных систем.

Можно посмотреть на проблему с другой стороны и считать, что мы располагаем некоторым универсальным множеством всевозможных систем, осуществляющих трансляции цепочек конкретного языка. В этом случае возникает интересный вопрос — можно ли, применив это множество к определенному входу, транслировать этот вход адаптивно, т. е. таким образом, чтобы часть его транслировалась одним способом, часть — другим, и сам факт выбора некоторого механизма для трансляции части входа регулировался самим же входом и уже транслированной частью. Видимо ближе всего к этому подошел язык программирования РЕФАЛ, несмотря на то, что он позиционируется как многофункциональный язык. РЕФАЛ содержит два весьма важных понятия: конкретизационные скобки и сопоставление с образцом. Действительно, динамический характер "блуждания" конкретизационных скобок по цепочке символов, а также тот факт, что одному и тому же формальному вхождению свободной переменной могут соответствовать разные значения (в одном и том же трансляционном "сеансе"), как раз и приближает нас к решению декларированной задачи об адаптивной трансляции.

Интересная задача может возникнуть при исследовании способов компактификации спецификации некоторых подклассов контекстно-свободных грамматик. Под *компактификацией* понимается способ записи грамматики таким образом, чтобы сам способ "подсказывал" вид порождаемого языка. Для одного частного случая контекстно-свободных грамматик, а именно для автоматных грамматик, эта задача уже решена и ответ на нее носит название "регулярные выражения". Следует принимать во внимание также и тот факт, что нереально решить эту задачу для всего класса контекстно-свободных грамматик. Таким образом, на передний план здесь выходят эвристические приближения к идеалу.

Во многих областях современного системного и прикладного программирования приходится иметь дело с такими хорошо структурированными, наиболее просто описывающимися входными данными, как язык, специфицируемый некоторой контекстно-свободной грамматикой. Было бы неразумно отказываться от возможности автоматической генерации трансляторов с такого класса языков. Однако проблема до сих пор в полной мере не решена. На сегодняшний день в мире не существует единого, универсального подхода к вопросу спецификации семантического значения для цепочек, принадлежащих к какому-либо языку, неважно, формальному или естественному.

4.6.4.1. Подходы к задаче синтаксического анализа

Математическая постановка задачи синтаксического анализа предельно проста и фактически представляет собой задачу спецификации соответствия между некоторыми двумя множествами формальных объектов. Однако существует огромная разница между подходами математика и программиста к описанию соответствий между такого рода объектами. Прежде всего, эту разницу можно объяснить тем, что для математика не так важна временная составляющая, ибо рассматриваемые им объекты существовали, существуют и будут существовать во времени независимо от того, каким образом они были определены. Таким образом, единственной проблемой может стать установление формальной идентичности двух определений одного и того же объекта. В случае программиста мы имеем несколько иную ситуацию, а именно: все формальные объекты, с которыми ему приходится работать, либо атомарны, либо создаются на основе атомарных объектов за вполне конечное время, причем обычно, крайне желательным является минимизирование количества времени, затрачиваемого на построение полной иерархии объектов, решающих ту или иную задачу.

Вообще говоря, процесс задания трансляции есть не что иное, как конструктивное описание некоторой функции, в общем случае обладающей свойством инъективности. С математической точки зрения для этого достаточно каким-либо образом описать подмножество (возможно бесконечное) декартового произведения двух множеств. С точки зрения программиста такой подход не только недопустим, но и зачастую невозможен, поскольку простое перечисление элементов такого подмножества вероятно лишь в случае его конечности. Таким образом, первоочередной проблемой становится решение задачи отыскания скрытой структуры, описывающей способ построения по каждому элементу первого множества его образа относительно интересующей нас функции во втором множестве. Иными словами, должен существовать некоторый "закон", согласно которому, рассмотрев структуру произвольного элемента, мы сможем найти структурно подобный ему образ во втором множестве. Фактически мы разбиваем задачу на две подзадачи:

- задачу нахождения внутренней структуры произвольного элемента. Она носит название задачи синтаксического анализа;
- задачу нахождения элемента (в большинстве интересных случаев единственного), строение которого описывается найденной структурой. Она носит название трансляции или, более точно, синтаксически управляемой трансляции.

Задача синтаксического анализа, в классическом понимании, — это процесс структурирования некоторого представления при помощи заранее заданной грамматики. Разумеется, существенным условием, накладываемым на грамматику, является условие конечности набора ее правил. Множество же представлений, напротив, чаще всего оказывается бесконечным. Мы не будем

углубляться здесь в теоретические и философские аспекты задачи синтаксического анализа, а лишь отметим, что методы, используемые для ее решения, вообще говоря, мажорируют методы синтаксически управляемой трансляции. То есть мы можем синтезировать образы лишь тех представлений, чью структуру мы в состоянии распознать. Отметим также, что на практике под грамматиками понимают грамматики в смысле Хомского, причем для ряда задач, связанных с формальными языками, оказывается достаточным ограничиться классом контекстно-свободных грамматик. Следствием такого ограничения является тот факт, что структура представлений носит ярко выраженный иерархический характер, наилучшим образом описываемый деревом вывода в заданной грамматике.

При введенных выше ограничениях задача трансляции превращается в проблему спецификации механизма, строящего некоторое представление по дереву вывода в заданной грамматике. Спецификация такого рода наиболее просто осуществляется при помощи привязки семантических атрибутов и способов их преобразования к каждому правилу грамматики. Таким образом, каждый узел дерева вывода оказывается снабженным собственным семантическим атрибутом, а под результатом трансляции мы понимаем семантический атрибут, приписанный корневой вершине. Данный подход носит название атрибутивных грамматик.

4.6.5. Язык спецификации грамматик

Приведем описание *языка спецификации грамматик* для системы, реализующей синтаксически управляемую трансляцию. Это система `yacc`, входной язык которой принадлежит к классу LR(1) или, более точно, LALR(1). Подробно система будет рассмотрена в *разд. 5.2.4.3*.

Входные данные для системы определяются одним или несколькими ASCII-файлами, содержащими спецификацию грамматики входного языка в расширенной формальной системе обозначений Бэкуса-Наура. Любой входной файл обязан иметь следующую структуру:

```
%{
    <секция операторов языка C>
%}
    <секция описаний>
%%
    <секция грамматических правил и семантических процедур>
%%
    <вторая секция операторов языка C>
```

Символы '`%%`' являются специальными синтаксическими разделителями.

Секция `<секция операторов языка C>` является необязательной. Она может содержать операторы типа `#include`, описания и т. п.

Секция `<секция описаний>` содержит лексемы, грамматические переменные, информацию о приоритетах и ассоциативности:

- `%token` — сообщает синтаксическому анализатору информацию о типе символов, возвращаемых лексическим анализатором;
- `%right` и `%left` — определяют ассоциативность и приоритет операций;
- `%prec` — присваивает операции соответствующий приоритет;
- `%type` — определяет тип элементов грамматики;
- `%union` — определяет тип стека.

Секция `<секция грамматических правил и семантических процедур>` представляет собой набор строчек, определяющих отдельные правила, которые могут включать семантические процедуры.

Секция `<вторая секция операторов языка C>` также является необязательной. Она может содержать операторы типа `main(){...; yyparse(); ...}`, `yylex(){...}` и др.

Входные данные преобразуются в функцию разбора с именем `yyparse()` и записываются в виде файла на языке программирования C. Далее эту функцию можно откомпилировать, связать с другими программами и выполнить.

4.7. Естественные языки

С точки зрения программиста *естественный язык* может быть интересен в следующих случаях:

- как язык постановки задачи для компьютера;
- как язык культурной среды программы с минимальными естественно-языковыми возможностями — для выдачи сообщений;
- как язык ведения диалога с программой.

4.7.1. Особенности естественных языков и культурных сред

Для познания нравов какого ни есть народа,
старайся прежде изучить его язык.
Пифагор

Различные страны мира применяют различные соглашения по написанию даты, времени, последовательности слов во фразах и т. п. Приведем некоторые примеры:

- время, которое в России обозначается 22:33, в Америке будет обозначаться как 10:33 p.m.;
- дата в России, обозначаемая как 22.07.2001, будет обозначаться в Америке как 07/22/2001.

Эти и другие особенности отражают культурную среду. Уже давно элементы культурной среды проникли в языки программирования. Например, стандарт языка программирования C вводит понятие "locale", которое является описанием культурной среды. По умолчанию программа будет выполняться в "культурной среде C". Стандартными функциями, имеющими специфику поведения в определенной культурной среде, являются: `atoi()`, `isalnum()`, `strtod()` и многие другие.

Естественный язык — это часть культурной среды. Укажем особенности естественных языков, создающие сложности при переносе программы из одной языковой среды в другую.

- *Разделители слов.* Обычно слова отделяются пробелами, однако ряд восточных языков не используют разделители между словами.
- *Порядок слов.* Порядок слов во фразах очень сильно отличается в различных языках. Порядок слов в выражении "черный кот" на русском языке должен быть изменен на противоположный в эквивалентном выражении на испанском языке — "uno gato Negro".
- *Порядок сортировки букв.* Каждый язык поддерживает определенный порядок букв в алфавите. При сравнении слов следует учитывать этот порядок.
- *Количество букв в алфавите.* Большинство европейских языков содержит около тридцати букв в алфавите. Некоторые восточные языки включают более 40 тысяч иероглифов.

Об эргономической стороне естественных языков

Проанализировав статистику боев с японцами в 1941—1945 годах, американцы обнаружили, что, несмотря на равенство сил, они побеждали чаще. Причину нашли в том, что в английском языке средняя длина слова — 5 букв, а в японском — 13. То есть пока японец объяснит, что к чему, американцы уже стреляют. После этого американцы стали давать короткие названия-клички как своим, так и чужим самолетам, кораблям и т. п. Когда эта информация была получена в СССР, то специалисты вычислили среднюю длину слова в русском языке — 7 букв. Однако в процессе управления боем командир автоматически переходит на мат, и информативность речи возрастает в 2-3 раза.

О политкорректности в программировании

Американская культурная среда озабочена проблемой политкорректности не только в сфере человеческих отношений, но и сфере, связанной с программированием. Приведем лишь два примера: предлагается заменить команду `man` командной строки операционной системы Unix на команду `person`, а системный вызов `abort()` на `choice()`.

Существуют основные стандарты на представление текста.

- Американский стандартный код для информационного обмена (American Standard Code for Information Interchange — ASCII) наиболее распространен

в мире. Стандартная таблица этого кода имеет довольно ограниченные возможности и позволяет хранить только буквы латинского алфавита. Настоящий первоначальный стандарт был семибитным.

- Европейская модификация ASCII называется Latin-1 и представляет собой восьмибитную кодовую таблицу, в верхней части которой (коды от 128 до 255) располагаются специфические буквы алфавитов европейских языков и символы псевдографики.

> ***Об обработке художественных текстов***
>
> Обратим внимание на то, что в кодировке Latin-1 невозможно обрабатывать тексты многих классических произведений русской литературы (например, роман "Война и мир"), поскольку они содержат фрагменты на французском и немецком языках.

- Unicode — универсальный стандарт для кодирования букв и символов. В нем применяется шестнадцатибитный набор кодов. В принципе, в нем можно закодировать до 65 536 символов. Под кириллицу (Россия, Украина, Беларусь, Болгария, Югославия и др.) в Unicode отведено 256 мест.

4.7.2. Семантический анализ естественных языков

Слова — это крючки, при помощи которых
мысли цепляются одна за другую.
Г. Бичер

Задача семантического анализа естественного языка возникла с появлением ЭВМ. Основоположником теории анализа можно считать Ноама Хомского (Noam Chomsky).

4.7.2.1. Подход Н. Хомского

Хомский применил теорию формальных грамматик к описанию естественного языка [Хомский 1962]. Он первый в явном виде сформулировал задачу превращения лингвистики в точную науку путем построения формализуемых моделей языка, которые могут быть реализованы на компьютере, назвав это *генеративной парадигмой*. До сих пор его теория трансформационной грамматики в различных видоизменениях является одной из наиболее популярных. Приведем этапы развития этой теории с начала 60-х годов XX века:

- период стандартной теории относится к началу 60-х годов XX века. Теория появилась для отражения проблем в представлении структур естественного языка с помощью контекстно-свободных правил;
- период расширенной стандартной теории относится к началу 70-х годов XX века. Система ограничений на правильность синтаксической структуры в этой теории задается независимыми правилами условий трансформаций;

- теория управления и связывания (Government and Binding — GB) была разработана в 80-х годах XX века;
- минималистская программа 90-х годов XX века, предполагала максимально возможное сокращение числа грамматических категорий и операций с ними. В этой программе также в явной форме сформулированы два основных принципа построения лингвистической модели по Хомскому:
  - концептуальная естественность (т. е. соответствие естественно-языковым феноменам);
  - независимая достоверность (это предполагает набор неформальных критериев, включающих простоту, неизбыточность и т. п.).

Целью лингвистического исследования Хомский ставит теорию грамматической компетенции носителя языка. Грамматические структуры описываются формально, при этом целью является развитие синтаксических формализмов, обеспечивающих точное описание множества предложений языка. Теория вводит три языковых уровня: семантический, синтаксический и фонологический. Основное внимание трансформационной грамматики уделяется синтаксическому уровню, который описывается автономно. По мнению Хомского, его теория описывает универсальную часть структуры языкового аппарата человека, не зависящую от конкретного языка. При этом предполагается наличие психологических оснований для этой теории, в частности, того, что такая универсальная грамматика является врожденной.

4.7.2.2. Подход Г. С. Цейтина

Семантические сети, разработанные как общий аппарат представления знаний, с самого начала активно использовались для построения систем обработки естественного языка.

Григорий Самуилович Цейтин [Цейтин 1985] предложил свою версию представления значения естественно-языковых выражений с помощью семантических сетей. Его ассоциативные сети содержат узлы, выражающие некоторые сущности (соответствующие объектам в тексте или во внешнем мире), и направленные дуги, соединяющие эти узлы. Содержимым узла может являться число, строка символов, процедура или конечное множество других узлов. Дуги именованы, и имена всех дуг, покидающих данный узел, должны быть различны, чаще всего они предназначены для обозначения семантических и синтаксических ролей. В частности, слова "открыть" и "закрыть" могут быть применены для обозначения действий с широким диапазоном объектов: открыть и закрыть дверь, бутылку, компьютерный файл. В терминах ассоциативных сетей эти слова также могут быть представлены как дуги, которые соединяют некоторые виды объектов и метод соответствующего действия. Фреймы также представимы в виде ассоциативной подсети, один узел которой обозначен как входной для фрейма. Порождение нового фрейма из его прототипа трактуется как копирование участка семантической сети. Важной операцией над сетью является слияние двух узлов: дуги с

одинаковыми именами при этой операции сливаются. Этот механизм был использован в нескольких проектах по анализу естественного языка в узкой предметной области, сетевое представление при этом играло роль механизма представления знаний, в частности, для систем автоматизированного проектирования, выбора оптимального пути и т. п.

4.7.2.3. Подход В. А. Тузова

Задача семантического анализа естественного языка очень сложна. Алгоритм такого анализа нетривиален, т. к. разным словам в разных контекстах диалога соответствует разный семантический смысл. Виталий Алексеевич Тузов [Тузов 1990] предложил использовать для представления текста семантический язык, адекватный русскому языку и пригодный для представления смысла текста в явном виде. Семантический словарь является переводным словарем с русского языка на семантический. Семантический язык содержит 72 базисные функции и около 600 базисных понятий. В основу создания семантического словаря положено три основных утверждения.

- Слово любого естественного языка — это имя функции `f(x1, x2, ..., xn)`, которая связывается с этим словом и называется его семантикой. Каждое конкретное значение слово получает только после подстановки конкретных значений.
- Предложение — единая законченная суперпозиция функций. Смысл предложения вычисляется в процессе выполнения суперпозиции.
- Речь (функционирование языка) — процесс синтеза и анализа предложений. Речь определяют слово, предложение и некоторая семантическая оболочка, которая осуществляет взаимодействие между семантикой и прагматикой языка.

4.7.3. Интернационализация и локализация программных продуктов

Границы моего языка — это границы моего мира.
Л. Витгенштейн

Для того чтобы обеспечить легкий перенос программных продуктов из одной культурной среды в другую, было предложено разрабатывать их с применением концепции интернационализации и затем выполнять локализацию.

Интернационализация — это способ разработки и написания программных продуктов, которые легко могут быть адаптированы для региональных рынков сбыта. Интернационализованные продукты не требуют переписывания перед тем, как они смогут работать с разнообразными языками и клиентами.

Существуют четыре уровня интернационализации, каждый следующий из которых включает требования всех предыдущих уровней и накладывает свои дополнительные требования [Solaris 1995].

- *Уровень шрифтов и кодировок*. Программное обеспечение удовлетворяет первому уровню интернационализации, если ни один из восьми битов, используемых для представления литеры, не используется по какому-либо дополнительному назначению. Такое программное обеспечение способно использовать кодировку Latin-1 (она же ISO 8859-1). Примером дополнительного использования может служить применение одного из битов для обозначения того, что данная литера отображается жирным шрифтом в текстовом редакторе.
- *Уровень форматов и сравнения строк*. Программное обеспечение удовлетворяет этому уровню, если его методы форматного вывода и сравнения строк допускают локализацию. С точки зрения разработки программ это означает, что, например, функция сравнения строк `strcmp()` должна быть заменена на `strcoll()`.
- *Уровень сообщений и представления текста*. Программное обеспечение удовлетворяет третьему уровню интернационализации, если весь видимый пользователем текст (сообщения об ошибках, текст на кнопках и иконках и т. п.) можно легко выделить из программы, например, собрав в отдельном файле. Текст в таком файле можно легко перевести на другой язык. Специальные функции (например, `catgets()`) должны выбирать файл, из которого будут браться сообщения.
- *Уровень поддержки азиатских языков*. Программное обеспечение удовлетворяет этому уровню, если оно поддерживает азиатские языки, которые требуют многобайтную (до 4-х байт) кодировку.

Локализация — процесс адаптации программных продуктов для выбранного региона со своим языком, традициями и культурной средой. Для того чтобы использовать локализованную версию программного продукта, достаточно изменить параметры настроек или переменную окружения. Например, в операционной системе Unix достаточно изменить переменную окружения `LANG`. Два возможных значения, которые может принимать эта переменная, таковы:

- `C` — для культурной среды C;
- `ru` — для русскоязычной культурной среды.

О сокращениях

Слова "интернационализация" и "локализация" являются достаточно длинными и в русском, и в английском языке. Программисты предложили следующий вариант сокращенной записи этих слов: "и17я" и "л9я" (соответственно "i18n" и "l10n" на английском). Сокращенный вариант образован первой и последней буквами слова и количеством букв между ними.

4.7.4. Отступление "о придании тексту эмоций"

511 16
5 20 337
712 19
2000047

Грустные стихи

Мы не можем слышать изменения голоса и видеть мимику лица автора при чтении написанного им текста. Наиболее актуальной эта проблема может быть в случае коротких сообщений, например, пересылаемых по электронной почте. Для того чтобы предотвратить недопонимание, были придуманы *улыбки* (*смайлики*). Их основная задача — передать эмоциональную и интонационную окрашенность высказывания. В табл. 4.4 приведены символьные коды некоторых наиболее распространенных улыбок с объяснением их смысла.

Таблица 4.4. *Наиболее распространенные улыбки*

Символьный код улыбки	Объяснение смысла улыбки
:-)	Ваша основная улыбка. Она используется для обозначения саркастического или шутливого утверждения
;-)	Подмигивающая улыбка. Пользователь только что сделал саркастическое замечание ("не бейте меня за то, что я только что сказал")
:-(	Недовольная "улыбка". Пользователь недоволен последним утверждением, или ему грустно
:-I	Равнодушная улыбка. Лучше, чем :-(, но не так хорошо, как :-)
:->	Пользователь только что сделал действительно острое саркастическое замечание. Хуже, чем ;-)
8:-)	Пользователь — маленькая девочка
:-)-8	Пользователь — большая девочка
:-{)	У пользователя усы
:-{}	У пользователя накрашены губы помадой
:'-(	Пользователь плачет
:'-)	Пользователь плачет от счастья
:-@	Пользователь кричит

Таблица 4.4 *(окончание)*

Символьный код улыбки	Объяснение смысла улыбки
:<)	Пользователь закончил престижное учебное заведение
:-&	У пользователя язык завязан узлом (изумление)
=:-)	Пользователь — глава семейства :-)
:-P	Пользователь показывает язык
:-O	Пользователь удивлен

Большинство улыбок можно использовать без носов (табл. 4.5).

Таблица 4.5. *Улыбки без носов*

Символьный код улыбки	Объяснение смысла улыбки
:)	Маленькая улыбка
:]	Дружеская маленькая улыбка того, кто с радостью станет вашим другом
:D	Смеется
:|	Равнодушен
:(	Печален
:O	Кричит
:,(	Плачет
:*	Целует

Эмоциональные улыбки представлены в табл. 4.6.

Таблица 4.6. *Эмоциональные улыбки*

Символьный код улыбки	Объяснение смысла улыбки
:-)	Ха-ха
:-(	Ууу...

Таблица 4.6 *(окончание)*

Символьный код улыбки	Объяснение смысла улыбки
|-)	Хи-хи
:-I	Хммм...
|-D	Хо-хо
:-O	Оо-паaа...
:->	Хе-хе
:-P	Ээээ...!

О развитии идеи улыбок

Идея смайликов была развита отечественными пользователями сети Интернет, предложившими большое количество графических аналогов некоторых слов ненормативной лексики (**http://www.debilarius.ru/slovar.htm**).

4.8. Заключение

4.8.1. Выводы и рекомендации

Дерни за веревочку, дитя мое, дверь и откроется!
Шарль Перро

Свободное владение несколькими языками программирования помогает программисту не меньше, чем свободное владение несколькими естественными языками любому специалисту (и, вообще говоря, любому человеку). Как минимум следует знать основной и вспомогательный (дополнительный) язык. Кстати, даже не очень глубокое знание вспомогательного языка может существенно повысить эффективность применения основного.

На выбор языка программирования влияют четыре основных фактора.

- Сравнительная пригодность языка программирования для данной задачи. Обзор языков, ориентированных на различные предметные области, был сделан в *разд. 4.5.2*.
- Избранная методология. Часто говорят, что язык поддерживает ту или иную методологию. Обычно это означает, что применение этого языка совместно с указанной методологией в совокупности дадут значительно больший эффект. Обзор языков, поддерживающих разные методологии, сделан в *гл. 2*.

- Степень важности для разработчика многочисленных характеристик и свойств, которые могут быть присущи или не присущи избираемому языку программирования. Подробно характеристики были рассмотрены в *разд. 4.1.1.3*.
- Степень знакомства программистов с языком программирования. Результаты исследований говорят о том, что производительность программиста, работавшего на некотором языке более трех лет, возрастает на треть по сравнению с программистом такого же уровня, но без опыта работы на данном языке [Boehm 1981]. Исследование специалистов компании IBM показало даже более существенные результаты. Программисты с длительным опытом программирования на некотором языке имеют производительность в три раза бóльшую, чем программисты с минимальным опытом программирования [Walston, Felix 1977].

4.8.2. Проблемы и перспективы развития

Через пять лет у нас будет один суперязык программирования,
только мы не можем установить начало
этого пятилетнего периода.
Алан Дж. Перлис

Тезисно сформулируем основные проблемы и перспективы языков программирования.

Во-первых, следует отметить, что новые языки программирования будут появляться и дальше. Рано или поздно у нас возникнут кардинально новые мысли о программировании, и, согласно гипотезе Хурфа-Сэпира, они потребуют новых языков.

Во-вторых, следует обратить внимание на грамотное воплощение идей, заложенных в языках. Идея может быть хорошей, даже отличной, но ее решение и воплощение в конкретном языке — неудачным и безобразным.

Некоторые выводы о распространении новых языков можно сделать на основе языка Java [Богатырев 1998].

- Продвижение языка Java происходило за счет мощной рекламы, а не из-за достоинств самого языка.
- На широкое распространение в наше время могут претендовать только те языки, которые поддержаны крупными фирмами, а не те, которые являются лучшими.
- Время распространения и становления языка программирования (для языков, получивших известность) составляет в среднем от 3 до 10 лет. Язык Java получил широкое распространение всего лишь за один год.

В-третьих, в условиях застоя в развитии языков программирования есть смысл тщательно проанализировать все полезные накопленные за это время идеи. Предпринимаются попытки создания единой семантики современных языков программирования, в каком-то смысле опять приводящие к идее "универсального" языка [Андреев 2001].

В-четвертых, Интернет требует языковых средств и подходов, обеспечивающих правильное взаимодействие большого числа независимо разработанных программ.

В-пятых, использование структур и данных, которые раньше бы считались неприемлемыми из-за их неэффективности, сейчас вполне допустимо и может привести к новой организации языков.

Общий, несколько обнадеживающий итог звучит так. Ряд известных фирм и компаний (например, Microsoft, Sun Microsystems и т. п.) постепенно включают в свои продукты идеи, предложенные несколько десятков лет назад Никлаусом Виртом.

4.8.3. Рекомендации по литературе

Горе тому, кто читает только одну книгу.
Джордж Герберт

Выделим следующие работы, посвященные анализу и обзору языков программирования.

- "Языки программирования. Практический сравнительный анализ" [Бен-Ари 2000]. Это один из лучших современных учебников по языкам программирования.
- "Основные концепции языков программирования" [Себеста 2001]. Книга содержит описание фундаментальных понятий программирования на примере вопросов разработки различных языковых конструкций.
- "Языки программирования: разработка и реализация" [Пратт, Зелковиц 2002]. Книга дает прекрасный подход к анализу языков на основе реализаторской позиции.
- "Универсальные языки программирования. Семантический подход" [Калинин, Мацкевич 1991]. В книге обобщены и систематизированы требования к современным языкам программирования.
- "Языки программирования: концепции и принципы" [Кауфман 1986]. Книга содержит основные принципы оценки, создания и использования современных языков программирования.
- "A Critique of C++ and Programming and Language Trends of the 1990s". Работа Яна Джойнера (Ian Joyner) (**http://www.literateprogramming.com/c++critique.pdf**) преставляет собой отличное введение в языки программирования и их сравнительный анализ.
- "Феномен науки: кибернетический подход к эволюции" [Турчин 2000]. Автор прослеживает эволюцию мира от простейших одноклеточных организмов до возникновения языка, мышления, развития науки и культуры.

Две следующие книги содержат информацию о языках моделирования.

- "Проектирование программного обеспечения экономических информационных систем" [Вендров 2000]. Книга дает представление о многих языках моделирования.
- "Самоучитель UML" [Леоненков 2001]. В книге содержится описание языка моделирования UML.

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают следующие темы:
  - моделирование данных;
  - переменные и типы данных;
  - механизмы абстракции;
  - системы типов;
  - семантика языков программирования;
  - разработка языков программирования.
- Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет следующие дисциплины:
  - информатика. Содержит раздел "языки программирования высокого уровня";
  - параллельное программирование;
  - функциональное программирование;
  - рекурсивно-логическое программирование.

4.8.4. Вопросы и задания

Спросить — минутный стыд, не спросить —
стыд на всю жизнь.

Японская мудрость

1. Почему программисту полезно иметь представление о разработке языков программирования, даже если он сам никогда не будет разрабатывать языки?
2. Что такое синтаксис и семантика языка программирования?
3. Какие аргументы можно привести "за" и какие "против" создания универсального языка для всех областей программирования?
4. Укажите доминирующие в настоящее время языки программирования и объясните причины их популярности.

5. В чем состоят современные тенденции развития языков программирования?
6. Чем определяется уровень языка программирования?
7. Перечислите основные характеристики языков программирования.
8. Что означает высказывание о том, что язык программирования является машинно-независимым?
9. Приведите пример недостатка ортогональности в одном из языков программирования (например, Pascal).
10. Приведите примеры языков, поддерживающих классические технологические процессы.
11. Зачем нужны определяющие методологии в программировании?
12. Приведите примеры языков, поддерживающих основные методологии программирования.
13. Каковы основные свойства объектно-ориентированных языков программирования?
14. Укажите особенности абстрагирования в информатике и перечислите три основные языковые абстракции.
15. Дайте определение типа данных и абстрактного типа данных.
16. Объясните, как механизмы абстракции поддерживают создание повторно используемых компонент программного обеспечения.
17. Укажите основные разновидности полиморфизма.
18. Что означает выражение "язык программирования со строгой типизацией"?
19. Покажите важность влияния различных моделей типизации на проектирование и использование языков программирования, а также на процесс трансляции.
20. Опишите различные свойства переменной, включая адрес переменной, значение, область видимости, "живучесть" и размер, а также объясните их смысл.
21. Укажите особенности структурного программирования и визуального структурного программирования.
22. В каких инструментах применяется анализ зависимости по управлению и по данным?
23. Каковы основные характеристики модульности?
24. Какие основные задачи решаются с помощью языков моделирования?
25. В чем отличие моделирования на основе структурной методологии от моделирования на основе объектно-ориентированной методологии?
26. Почему соответствие единому стилю программирования является важной технологической задачей, возникающей в процессе программирования?

27. Каковы основные принципы защитного программирования и как их следует применять на практике?
28. Какими критериями следует пользоваться при выборе языка программирования?
29. Выполните краткий обзор языков программирования, ориентированных на предметную область.
30. Каковы основные методы синтаксического разбора?
31. Напишите на языке программирования (например, Pascal) программу, корректную с точки зрения синтаксических определений, но некорректную с точки зрения компилятора этого языка.
32. Каковы основные особенности естественных языков и культурных сред?
33. Укажите последовательность задач, решаемых при интернационализации программных продуктов.

ЛИТЕРАТУРА

Как разнообразны, новы, интересны стали бы книги,
когда бы их авторы писали только то, что думают!
Л. Вовенарг

1. *Агафонов 1982* — Агафонов В. Н. Типы и абстракция данных в языках программирования // Данные в языках программирования. — М.: Мир, 1982.
2. *Агафонов 1984* — Агафонов В. Н. Языки и средства спецификации программ (обзор) // Требования и спецификации в разработке программ. — М.: Мир, 1984.
3. *Андреев 2001* — Андрей Андреев. Эволюция современных языков программирования // Мир ПК, № 3, 2001
4. *Ахо, Ульман 1978* — Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х т. — М.: Мир, 1978.
5. *Ахтырченко, Леонтьев 2001* — Ахтырченко К. В., Леонтьев В. В. Моделирование программной архитектуры // Вычислительные методы и программирование, 2001. Т. 2, с. 33—48.
6. *Баурн 1986* — Баурн С. Операционная система Unix. — М.: Мир, 1986.
7. *Бен-Ари 2000* — М. Бен-Ари. Языки программирования. Практический сравнительный анализ — М.: Мир, 2000.
8. *Богатырев 1998* — Богатырев Руслан. Гадание на кофейной гуще // Мир ПК, № 2, 1998, с. 120—133.
9. *Буч 1998* — Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. — М.: Бином, СПб.: Невский диалект, 1998.
10. *Буч, Рамбо, Джекобсон 2000* — Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. — М.: ДМК, 1998.
11. *Васкевич 1996* — Васкевич Д. Стратегии клиент/сервер. Руководство по выживанию для специалистов по реорганизации бизнеса. — К.: Диалектика, 1996.

12. *Вельбицкий 1984* — Вельбицкий И. В. Технология программирования. — Киев: Техніка, 1984.

13. *Вирт 1996* — Никлаус Вирт. Долой "жирные" программы // Открытые системы, № 6, 1996.

14. *Гослинг 1997* — Гослинг Д., Арнольд К. Язык программирования Java. — СПб.: Питер, 1999.

15. *Грановская, Березная 1991* — Грановская Р. М., Березная И. Я. Интуиция и искусственный интеллект. — Л.: Изд-во ЛГУ, 1991.

16. *Дейкстра 1975* — Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975, с. 7—97.

17. *Дейкстра 1978* — Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.

18. *Дейт 2000* — Дейт, К., Дж. Введение в системы баз данных, 6-е издание. — К.; М.; СПб.: Издательский дом "Вильямс", 2000.

19. *Деметрович, Кнут, Радо 1989* — Деметрович Я., Кнут Е., Радо П. Автоматизированные методы спецификации. — М.: Мир, 1989.

20. *Джоунз 1989* — Джоунз Г. Программирование на языке Оккам. — М.: Мир, 1989.

21. *Достоевский 1985* — Достоевский Ф. М. Преступление и наказание. — М.: Мир, 1985.

22. *Евграфов, Евграфов 1993* — Евграфов М. А., Евграфов Л. М. TeX: руководство по набору и редактированию математических текстов. — М.: Физматлит, 1993.

23. *Замулин 1990* — Замулин А. В. Типы данных в языках программирования и базах данных. — Новосибирск: Наука, 1990.

24. *Зиглер 1985* — Зиглер К. Методы проектирования программных систем. — М.: Мир, 1985.

25. *Калинин, Мацкевич 1991* — Калинин А. Г., Мацкевич И. В. Универсальные языки программирования. Семантический подход. — М.: Радио и связь, 1991.

26. *Кауфман 1986* — Кауфман В. Ш. Языки программирования: концепции и принципы. — М.: Изд-во МГУ, 1986.

27. *Кахро, Калья, Тыугу 1988* — Кахро М. И., Калья А. П., Тыугу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). — М.: Финансы и статистика, 1988.

28. *Кнут 1993* — Дональд Кнут. Все про TEX. — Протвино: АО RDTEX, 1993.

29. *Кун 1975* — Кун Т. Структура научных революций. — М.: Наука, 1975.

30. *Лавров 1986* — Лавров С. С. Архитектура баз знаний // Программное обеспечение вычислительных комплексов новой архитектуры. — Новосибирск: Изд-во СО ИТМ и ВТ, 1986, с. 3—13.

31. *Лейн 1991* — Лейн А. Объектно-ориентированное программирование в реальном мире // Мир ПК, 1990, № 5, с. 29—35.

32. *Лисков 1989* — Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ. — М.: Мир, 1989.

33. *Леоненков 2001* — Леоненков А. В. Самоучитель UML. — СПб.: БХВ-Петербург, 2001.

34. *Лукин 1999* — Лукин В. А. Художественный текст: основы лингвистической теории и элементы анализа. — М.: Ось-89, 1999.

35. *Майерс 1980* — Майерс Г. Надежность программного обеспечения. — М.: Мир, 1980.

36. *Мейерс 2000* — Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. — М.: ДМК, 2000.

37. *Налимов 1979* — Налимов В. В. Вероятностная модель языка. — М.: Наука, 1979.

38. *Паронджанов 1999* — Паронджанов В. Д. Как улучшить работу ума. — М.: Радио и связь, 1999.

39. *Попов 1987* — Попов Э. В. Экспертные системы. — М.: Наука, 1987.

40. *Пратт, Зелковиц 2002* — Пратт Т., Зелковиц М. Языки программирования: разработка и реализация. 4-е издание. — СПб.: Питер, 2002.

41. *Романовский 1999* — Романовский И. В. Дискретный анализ. — СПб.: Невский диалект, 1999.

42. *Сафонов 1989* — Сафонов В. О. Языки и методы программирования в системе "Эльбрус". — М.: Наука, 1989.

43. *Себеста 2001* — Себеста Р. У. Основные концепции языков программирования, 5-е изд. — М.: Издательский дом "Вильямс", 2001.

44. *Симкин 1996* — Симкин Стив, Бартлетт Нейл, Лесли Алекс. Программирование на Java. Путеводитель. — К.: НИПФ "ДиаСофт Лтд.", 1996.

45. *Страуструп 2000* — Страуструп Б. Дизайн и эволюция C++. — М.: ДМК Пресс, 2000.

46. *Тузов 1990* — Тузов В. А. Языки представления знаний. — Л.: Изд-во ЛГУ, 1990.

47. *Турский 1981* — Турский В. Методология программирования. — М.: Мир, 1981.

48. *Турчин 2000* — Турчин В. Ф. Феномен науки: кибернетический подход к эволюции. — М.: ЭТС, 2000.

49. *Филлипс, Гарсиа-Диас 1984* — Филлипс Д., Гарсиа-Диас А. Методы анализа сетей. — М.: Мир, 1984.

50. *Фитиалов 1984* — Фитиалов С. Я. Формальные грамматики. — Л.: Изд-во ЛГУ, 1984.

51. *Фокс 1985* — Дж. Фокс. Программное обеспечение и его разработка. — М.: Мир, 1985.

52. *Хоар 1975* — Хоар Ч. О структурной организации данных // Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975.

53. *Хомский 1962* — Хомский Н. Синтаксические структуры // Новое в лингвистике. Вып. 11. — М.: Иностранная литература, 1962, с. 412—527.

54. *Цейтин 1985* — Цейтин Г. С. Программирование на ассоциативных сетях // ЭВМ в проектировании и производстве, вып. 2. — Л.: Машиностроение, 1985.

55. *Цейтин 1990* — Цейтин Г. С. На пути к сборочному программированию // Программирование. — 1990. — № 1, с. 78—92.

56. *Шлеер, Меллор 1993* — Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. — Киев: Диалектика, 1993.

57. *Языки 1989* — Языки программирования Ада, Си, Паскаль. Сравнение и оценка. — М.: Радио и связь, 1989.

58. *Appleby 1991* — Doris Appleby. Programming Languages: Paradigm and Practice. McGraw-Hill, 1991.

59. *Bal, Grune 1994* — Henri E. Bal, Dick Grune. Programming Language Essentials. Addison-Wesley Publishing Company, 1994.

60. *Boehm 1981* — Boehm B. W. Software Engineering Economics. Prentice Hall, 1981.

61. *Cardelli, Wegner 1985* — Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polimorphism // Computing Surveys, Vol 17, N 4, pp. 471—522, December 1985.

62. *Grune, Jacobs 1998* — Dick Grune, Ceriel J. H. Jacobs. 1998.

63. *Handbook 1998* — Handbook of Programming Languages. Macmillian Technical Publishing, 1998.

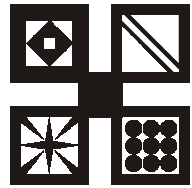
64. *Holt 1975* — R. C. Holt. Structure of Computer Programs: A Survey // Proceedings of the IEEE, pp. 879—893, 63(6), 1975.

65. *Morris 1973* — J. H. Morris, Jr. Types are not sets // Conference Record of the ACM Symposium on Principles of Programming Languages, pp. 120—124, Boston, October 1973.

66. *Solaris 1995* — Solaris 2.5. Developer's Guide to Internationalization. SunSoft Press, 1995.

67. *Terekhov, Verhoef 2000* — Andrey A. Terekhov, Chris Verhoef. The Realities of Language Conversions // IEEE Software, November/December 2000.

68. *Walston, Felix 1977* — Walston C. E., Felix C. P. A Method of Programming Measurement and Estimation // IBM Systems Journal, Vol. 16, No. 1, 1977, pp. 54—73.



Глава 5

Системы программирования

В этой главе мы подробно поговорим о нескольких группах инструментов, которые относятся к системам программирования.

- Инструментарий поддержки технологических процессов. Мы выделим инструменты, ориентированные на один процесс и применяемые только в нем, а также универсальные (независимые от процесса).
- Инструментальные системы (инструментальные среды программирования, средства автоматизации разработки программ, интегрированные среды и репозитории проекта).
- Инструментарий поддержки коллективной разработки.

Глава включает обзор систем поддержки естественно-языкового интерфейса.

5.1. Введение в системы программирования

5.1.1. Основные понятия и определения

Еще не все вещи имеют свои имена.

Габриель Гарсиа Маркес

Система программирования — это часть базового программного обеспечения, поддерживающая процесс программирования и работу готовых программ. Системы программирования представляют собой единство средств статической (инструментальной) и динамической (исполнительной) поддержки.

К настоящему моменту сложилось представление о традиционном составе системы программирования, в который входят следующие программные инструменты и библиотеки:

- редактор для построения программ;
- транслятор для перевода программ с языка программирования на машинный язык;

- отладчик для проверочных запусков программ и исправления ошибок;
- библиотеки периода трансляции и периода исполнения;
- средства управления компиляцией и построением программного проекта;
- монитор, интегрирующий в себе вышеперечисленные средства и организующий функционирование системы программирования в целом.

Этот минимум, позволяющий достаточно эффективно вести процесс создания программ, был представлен еще в начале 70-х годов XX века. Однако настоящую популярность такой интегрированный набор инструментов приобрел лишь в середине 1980-х годов благодаря компании Borland Software Corporation (**http://www.borland.com/**). Пакет Turbo Pascal впервые появился в 1983 году для операционной системы CP/M, а в начале 1984 года был перенесен в MS-DOS. Практически все начало истории персональных компьютеров IBM PC связано с системой Turbo Pascal — наиболее популярной средой разработок.

Программный инструмент — это программа, предназначенная для поддержки разработки программных продуктов. Например, отладчик, облегчающий программисту выполнение отладки продукта.

О широкой трактовке понятия "инструмент"

В принципе, любую программу можно считать "инструментом", а набор программ — "инструментарием". В этой главе мы будем пользоваться данным выше определением программного инструмента, указывая на задачу, которую он решает. Термин "инструментарий", как правило, мы не будем употреблять, заменяя его "системой".

Утилита — программа, обеспечивающая некоторые общие функции (например, копирование файлов, подготовку текстов, организацию перекрестных ссылок и т. п.).

Библиотеки процедур — наборы процедур различного назначения, упрощающие разработку приложений. Например, библиотека функций ввода-вывода или математических функций.

Программы предоставления дополнительных услуг — это программы, предлагающие необязательные, но полезные функции (например, калькулятор или надстройка над полезной утилитой, реализующая графический пользовательский интерфейс к ней).

О связи систем программирования и операционных систем

Очень часто многие модули операционных систем используются в качестве компонентов систем программирования.

5.1.2. История и эволюция

5.1.2.1. Некоторые важные даты

Существуют два лучших времени, чтобы посадить дерево:
первое — 20 лет назад, а второе — сейчас.
Восточная мудрость

Приведем некоторые вехи истории систем программирования.

- 1949 г. — Джон Мочли разработал примитивный интерпретатор для своего языка "Short Code".
- 1952 г. — Грейс Мюррей Хоппер (Grace Murray Hopper) разработала "A-O" — первый компилятор. Компилятор осуществлял функцию объединения и в ходе трансляции производил организацию подпрограмм, выделение памяти компьютера и преобразование псевдокода в машинные команды. Происхождение введенного ею термина "компилятор" она объясняла тем, что подбор материала из библиотеки подпрограмм называется компиляцией.
- 1955 г. — А. П. Ершов с группой программистов создал программирующую программу (транслятор) для БЭСМ. В 1958 г. им была опубликована монография "Программирующая программа для быстродействующей электронной счетной машины", которая является первой в мировой литературе монографией по автоматизации программирования.
- 1957 г. — Джон Бэкус (John Backus) и его сотрудники установили в IBM первый компилятор языка FORTRAN.
- 1962 г. — научным коллективом под руководством С. С. Лаврова и В. А. Степанова (отдел динамики полета в КБ С. П. Королева, г. Калининград Московской области) разработан отечественный транслятор ТА-1 для международного языка программирования Algol-60. Несколько позже были разработаны трансляторы ТА-2 (разработка отдела прикладной математики МИАН под руководством М. Р. Шура-Буры и Э. З. Любимского, Москва) и Альфа-транслятор (разработка группы А. П. Ершова, ВЦ СО АН СССР, Новосибирск). Эти трансляторы открыли эру практического использования языков программирования высокого уровня в СССР.
- 1972 г. — Г. С. Цейтин, С. Н. Баранов и А. Н. Терехов в Ленинградском университете разработали DICO — интерактивный текстовый редактор для ЭВМ ODRA 1204 с консольной пишущей машинкой.
- 1979 г. — Г. Ф. Дейкало и Б. А. Новиков в Ленинградском университете начали создавать систему JEC — интерактивное многопользовательское расширение IBM OS/360-OS/370, включавшее разработку компонент интерфейса с операционной системой, задачи взаимодействия с пользователем, взаимодействие с консольным оператором.

Среди отечественных программистов особой популярностью пользуется система программирования на базе языка программирования Pascal компании Borland Software Corporation. Сама компания была основана в 1983 году французским математиком Филиппом Каном (Philippe Kahn). Приведем некоторые даты, связанные с развитием этой системы.

- 1983 г. — Создание Андерсом Хейлсбергом (Anders Hejlsberg) программы Turbo Pascal для операционной системы CP/M.
- 1984 г. — Перенос Turbo Pascal в MS-DOS.
- 1990 г. — Представление Turbo Pascal для Windows.
- 1992 г. — Последний выпуск Borland Pascal версии 7.0.
- 1995 г. — Появление Delphi — доведенной до логического совершенства и кульминационного пика интегрированной среды Turbo Pascal.

5.1.2.2. Основные этапы в формировании состава систем программирования

Можно выделить три основных этапа в формировании состава систем программирования.

- Для начала 70-х годов XX века было характерно наличие "монолитных" систем программирования, состоящих из одной-двух больших программ, выполняющих множество разнообразных функций, начиная с редактирования и трансляции исходного модуля и заканчивая некоторыми сервисными возможностями.
- По мере развития отдельных функций программы происходит их выделение в самостоятельные компоненты, такие как текстовый редактор, отладчик, загрузчик.
- Следующим шагом явилось распространение инструментальной поддержки на все технологические процессы жизненного цикла программ.

5.1.3. Классификация

5.1.3.1. Классификация по ориентации на поддержку процессов

Начнем с основной классификации систем программирования, которая положена в основу структуры данной главы.

- *Инструментарий поддержки технологических процессов*:
  - инструментарий, поддерживающий определенный технологический процесс (процесс-ориентированный инструментарий);
  - универсальный инструментарий, который можно использовать в нескольких технологических процессах (процесс-независимый инструментарий).

- *Инструментальные системы разработки и сопровождения программных продуктов*:
  - инструментальные среды программирования;
  - средства автоматизации разработки программ (CASE-средства);
  - интегрированные среды и репозитории проекта.
- *Инструментарий поддержки коллективной разработки.*

5.1.3.2. Функциональная классификация

Еще одну классификацию систем программирования можно провести, отражая их *функциональную ориентацию* в технологическом процессе. Эта классификация относится к группе инструментария поддержки технологических процессов.

Рассмотрим четыре группы инструментов [Жоголев 1996].

- *Редакторы*, поддерживающие конструирование (формирование) многочисленных программных документов.
- *Анализаторы*, производящие:
  - статическую обработку документов, осуществляя различные виды их контроля, выявление определенных свойств и накопление статистических данных;
  - динамический анализ программ.
- *Преобразователи*, позволяющие:
  - автоматически приводить документы к другой форме представления;
  - синтезировать какой-либо документ из отдельных частей.
- Инструменты *поддержки процесса выполнения программы*, позволяющие исполнять:
  - описания процессов или отдельных их частей, представленных в виде, отличном от машинного кода;
  - машинный код с дополнительными возможностями его интерпретации;
  - обращение к библиотекам подпрограмм (модулей) и базам данных.

5.1.3.3. Классификация по категориям

Классификация систем программирования *по категориям* [Калянов 1996] определяет уровень их интегрированности по выполняемым функциям и включает:

- *вспомогательные программы* (tools) — пакеты, решающие небольшую автономную задачу, принадлежащую проблеме более широкого масштаба;

- *пакеты разработчика* (toolkit) — совокупность интегрированных программных средств, обеспечивающих помощь для одного из классов программных задач;
- *инструментальные средства* (workbench) — интеграция программных средств, которые поддерживают системный анализ, проектирование и разработку программного обеспечения. При этом используется репозиторий, содержащий всю техническую и управляющую информацию о проекте. Иногда эту категорию систем называют рабочими средами разработчика (environment).

5.1.3.4. Классификация по предоставляемому интерфейсу

Системы программирования предоставляют два ярко выраженных типа программного интерфейса.

- *Интерфейс командной строки* (Command Line Interface — CLI). Это традиционный интерфейс систем программирования в операционной системе Unix. В современных диалектах Unix практически все инструменты имеют также надстройку с графическим пользовательским интерфейсом.
- *Графический пользовательский интерфейс* (Graphic User Interface — GUI). Этот интерфейс традиционен для систем программирования в Windows.

5.2. Процесс-ориентированный инструментарий

Всякий инструмент будет наилучшим образом удовлетворять своему назначению, если он предназначен для исполнения одной работы, а не многих.

Аристотель

Теперь подробно перечислим инструменты, относя их к одной из следующих групп.

5.2.1. Возникновение и исследование идеи

Вдохновение нужно в поэзии, как и в геометрии.

Александр Сергеевич Пушкин

Для поддержки данного процесса предлагается использовать два типа инструментов.

- *Поисковые информационные системы.* Эти системы по сделанному запросу выдают ссылки на найденные документы. Подробно такие системы будут рассмотрены в *разд. 5.3.6.* Наиболее известными поисковыми системами являются:
  - поисковые системы общего назначения (например, Яndex (**http://www.yandex.ru**) и Google (**http://www.google.com**));

- тематические каталоги Интернета (например, Yahoo (**http://www.yahoo.com**));
- базы данных патентов в Интернете (например, Questel-Orbit (**http://www.questel.orbit.com**));
- индексы цитирования научных статей (например, ResearchIndex (**http://citeseer.ist.psu.edu/cis**)).

❒ *Системы поиска решений*. Эти системы представляют собой структурированный набор разнообразного рода эвристических приемов и методов (**http://www.altshuller.ru/triz/**). Они объединяют предшествующий опыт различных авторов и коллективов и должны способствовать увеличению количества рождающихся идей.

5.2.2. Управление

Сильный действует рукой, мудрый — умом, а хитрый — кем-то еще.

Владислав Гжегорчик

Инструментарий поддержки процесса управления разделим на шесть основных групп.

❒ Системы управления проектами.

❒ Простейшие организационные средства:
- электронная почта;
- электронный календарь.

❒ Виртуальный офис проекта.

❒ Средства оценки качества.

❒ Хранилища данных:
- корпоративные хранилища данных;
- витрины (киоски) данных (например, интранет).

❒ Системы поддержки принятия решений:
- информационные системы руководителя (статические);
- оперативной аналитической обработки (динамические).

5.2.2.1. Системы управления проектами

Перечислим группы средств, предназначенных для управления проектами:

❒ средства описания комплекса работ проекта, связей между отдельными работами и их временных характеристик;

❒ средства поддержки информации о ресурсах и затратах по проекту и его отдельным работам;

- средства контроля над ходом выполнения проекта;
- графические средства представления структуры проекта и средства создания отчетов по проекту.

Система должна позволять динамически пересматривать планы и сроки работ в ходе выполнения проекта. Возможно, система будет иметь средства отслеживания назначений сотрудников.

Подробную информацию о системах управления проектами, включающую описание их характеристик, можно получить в Интернете (**http://www.aproject.ru/tools/**). Перечислим ряд наиболее интересных систем:

- Microsoft Project (компании Microsoft (**http://www.microsoft.com/rus/office/Default.mspx**)) — самая распространенная в мире на сегодняшний день система, которая может быть рекомендована для планирования несложных проектов. Пакет достаточно прост, имеет традиционные средства помощи и удобные инструменты создания отчетов. Расширенное семейство продуктов Microsoft Project 2002 сочетает в себе понятные средства управления проектами, доступа к информации и поддержки коллективной работы, а также является мощной платформой, способствующей повышению производительности труда и более эффективному ведению бизнеса;
- система Time Line (компании Time Line, Inc. (**http://www.timeline.com**)) также предназначена для несложных проектов. Содержит минимально необходимые функции управления проектами;
- система Primavera (компании Primavera Systems, Inc. (**http://www.primavera.com**)) — более мощная система, применяемая для управления средними и крупными проектами.

5.2.2.2. Простейшие организационные средства

Электронная почта может играть важную организационную роль в процессе управления проектом. Достоинства такого подхода в том, что вся информация пересылается по почте и остается "в истории". Во многих компаниях принято подкреплять почтой решения, достигнутые в результате устной договоренности. В крупных компьютерных компаниях инженер, как правило, получает по электронной почте около 100 рабочих писем в день, а менеджер — от 300 до 500. В настоящее время набор почтовых программ входит в состав большинства операционных систем.

Электронный календарь представляет собой информационный пакет, в который заносятся пометки о будущих событиях, таких как совещания, семинары, контрольные сроки важнейших событий. Информация может быть занесена как самостоятельно исполнителем, так и другими людьми, которым разрешен доступ (например, секретарем). Электронные календари реализованы в графических оболочках большинства современных операционных систем.

5.2.2.3. Виртуальный офис проекта

Виртуальный офис проекта — это средство организации групповой работы над проектами на базе Web-технологий и с использованием Интернета. Офис предоставляет удаленный доступ к данным проектов (например, систем Open Plan и Microsoft Project), взаимодействие участников проекта в рамках единого портала проекта, управление документами проекта, а также регламентирует ввод прогресса работ исполнителями.

Пример виртуального офиса — система WelcomHome (компании Welcom Software). Она облегчает совместную работу на уровне проекта и группы проектов, обеспечивая единое хранилище информации о достижении целей проекта, о ходе исполнения проекта и возникающих рисках. Для упрощения системы коммуникаций и своевременного осведомления о ходе проекта всех заинтересованных сторон предусмотрены следующие возможности: электронная почта, доски объявлений и дискуссионные форумы.

5.2.2.4. Средства оценки качества

Для сравнения качества программных продуктов применяются количественные методы оценки. Среди программ оценки качества отметим Metricate (компании Software Productivity Centre (**http://www.spc.ca**)), которая анализирует все аспекты деятельности компаний по производству программного обеспечения: эффективность технологических процессов, качество программного кода, уровень управления проектами, стоимость выполнения различных этапов, производительность получаемой системы, продуктивность труда разработчиков и качество готовых изделий.

5.2.2.5. Хранилища данных

Для того чтобы обеспечить возможность анализа накопленных данных, организации стали создавать *хранилища данных* (data warehouse), которые представляют собой интегрированные коллекции данных, собранных из различных систем оперативного доступа к данным.

Корпоративные хранилища данных содержат информацию, относящуюся ко всей корпорации и собранную из множества оперативных источников для консолидированного анализа. Корпоративное хранилище содержит детальную и обобщающую информацию, его объем может составлять от 50 гигабайт до одного или нескольких терабайт.

Витрины (киоски) данных содержат подмножество корпоративных данных и строятся для подразделений внутри организации. Киоски данных часто строятся силами самого подразделения и охватывают конкретный аспект, интересующий его сотрудников. Витрина данных может получать данные из корпоративного хранилища (зависимая витрина), или, что более распространено, данные могут поступать непосредственно из оперативных источ-

ников (независимая витрина). Одна из разновидностей витрин — *интранет*. Этим термином обозначают корпоративные информационные системы, построенные на принципах, заимствованных из сети Интернет. По сути, интранет — это перенос хорошо известных сетевых технологий во внутрикорпоративные сети. Внутренняя сеть позволяет создать целостный портрет организации и обрести радикально новую систему внутренних коммуникаций и доступа к информационным ресурсам. Интранет может быть значительно "умнее" Интернета, потому что он контролируем, и в него можно инвестировать значительно больше знаний и средств. В качестве серверов интранета можно использовать хорошо известные серверы Интернета, например Apache (**http://www.apache.org**).

5.2.2.6. Системы поддержки принятия решений

Информационные системы руководителя (Executive Information Systems) содержат в себе предопределенные множества запросов и, будучи достаточными для повседневного обзора, неспособны ответить на все вопросы к имеющимся данным, которые могут возникнуть при принятии решений. Результатом работы такой системы, как правило, являются многостраничные отчеты, после тщательного изучения которых у аналитика появляется новая серия вопросов. Однако каждый новый запрос, не предусмотренный при проектировании такой системы, должен быть сначала формально описан, закодирован программистом и только затем выполнен.

Динамические системы поддержки принятия решений, напротив, ориентированы на обработку нерегламентированных (ad hoc) запросов аналитиков к данным. Наиболее глубоко требования к таким системам сформулированы в работе [Codd, Codd, Salley 1993], положившей начало концепции OLAP (On-Line Analytical Processing). Работа аналитиков с такими системами заключается в формировании интерактивной последовательности запросов и изучении их результатов. В основе концепции OLAP лежит принцип многомерного представления данных.

5.2.3. Анализ требований и проектирование

5.2.3.1. Системы на основе структурной методологии

Для анализа требований и проектирования на основе структурной методологии могут быть применены следующие системы:

- Silverrun ModelSphere (компании magma solutions GmbH (**http://www.silverrun.com/front.html**)) поддерживает методы DATARUN, Гейна-Сарсона, Йордона, Мартина и др.;

- Oracle Designer (компании Oracle (**http://www.oracle.com**)) поддерживает CASE*Method Баркера;
- ERwin (компании Computer Associates International, Inc. (**http://ca.com/**)) поддерживает диаграммы функционального моделирования.

> ***О системах на основе структурной методологии***
>
> Обратим внимание, что практически все перечисленные системы в настоящее время поддерживают также и объектно-ориентированную методологию.

Из отечественных систем укажем следующие:

- ДРАКОН, поддерживающая ДРАКОН-схемы [Пароннджанов 1999];
- CASE.Аналитик (компании Эйтекс), поддерживающая подход Гейна-Сарсона. Система работает с иерархией диаграмм, последовательно детализирующих модель (рис. 5.1).

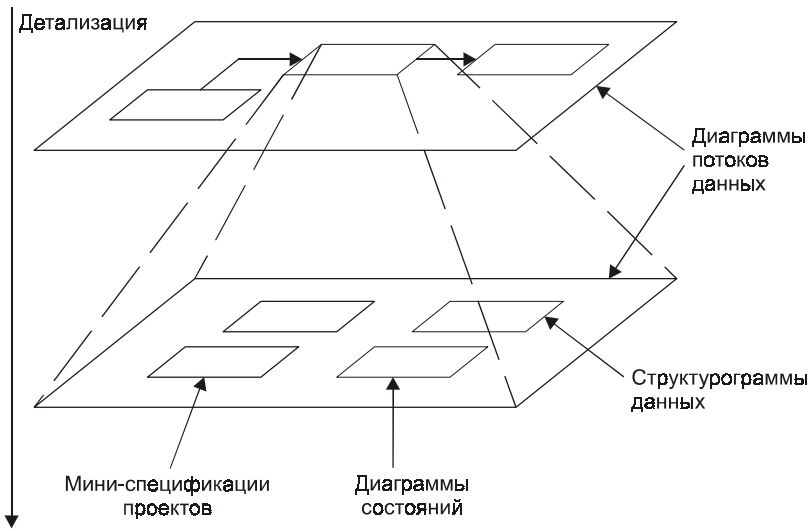


Рис. 5.1. Иерархия диаграмм в системе CASE.Аналитик

В состав системы CASE.Аналитик входят [Калянов 1996]:

- база данных проекта;
- графические редакторы потоковых диаграмм и структурограмм;
- средства вывода экранных и печатных форм;
- документатор;
- верификатор, позволяющий вести автоматический контроль выполнения формальных правил построения модели при вводе и редактировании.

5.2.3.2. Системы на основе объектно-ориентированной методологии

Укажем лучшие и наиболее известные системы для анализа требований и проектирования на основе объектно-ориентированной методологии:

- Rational Rose (компании Rational Software Corporation (**http://www-306.ibm.com/software/rational/**));
- Together Control Center (компании TogetherSoft Corporation (**http://www.togethersoft.com**)).

Перечислим основные достоинства и возможности системы Rational Rose:

- система прошла достаточно долгий путь развития и совершенствования. Она поддерживает язык UML и ряд ранних языковых нотаций (например, Booch и OMT (Object Modeling Technique));
- система реализована на обеих наиболее распространенных операционных системах: Unix и Windows;
- система имеет три основные модификации. Пользователь имеет возможность выбрать одну из них, устраивающую его по финансовым соображениям:
  - Enterprise — с возможностью генерации кода на языках Visual C++, Visual BASIC, Java, CORBA IDL. Еще совсем недавно существовала поддержка языка Smalltalk;
  - Professional — возможность генерации кода на одном из перечисленных языков;
  - Modeler — без языковой поддержки;
- система поддерживает восстановление спецификаций из кода (при очень серьезных ограничениях на то и другое);
- система поддерживает генерацию проектной документации.

5.2.4. Программирование (реализация)

5.2.4.1. Трансляторы

Диван был транслятором. Он создавал вокруг себя поле, преобразующее, говоря просто, реальность действительную в реальность сказочную.

Аркадий и Борис Стругацкие. "Понедельник начинается в субботу"

Транслятор — программный инструмент, предназначенный для перевода (трансляции) программ с одного формального языка на другой. Существует несколько основных видов трансляторов:

- компиляторы;
- декомпиляторы;
- интерпретаторы.

Общая схема трансляторов, компонентов трансляторов и связанных с ними инструментов представлена на рис. 5.2, где сплошная линия означает обязательную связь между компонентами, а пунктирная — возможность включения одного компонента в другой. Подчеркнем, что многие компоненты могут являться общими для различных видов трансляторов и инструментов.

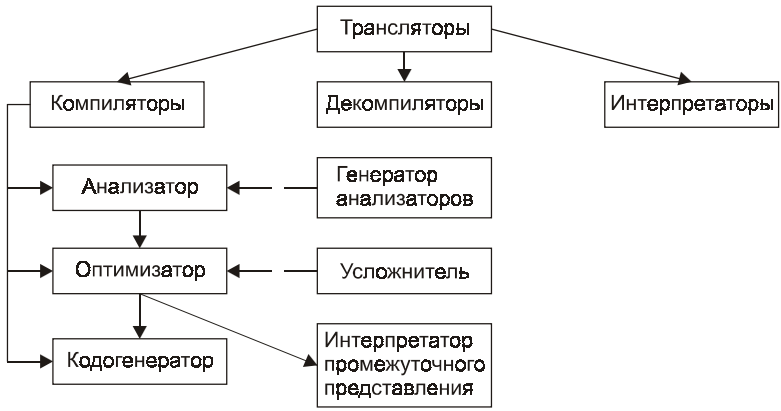


Рис. 5.2. Трансляторы

Обратим внимание на то, что многие другие программные инструменты могут рассматриваться как языковые трансляторы, например:

- текстовые редакторы (транслируют текст и специфические команды форматирования в некоторый образ);
- процессоры запросов (транслируют высокоуровневый язык запросов в реляционный язык, например SQL);
- программы доказательства теорем (преобразуют спецификацию теоремы в некоторый метаязык).

5.2.4.2. Компиляторы

Компиляторы в некотором смысле подобны зубным врачам: они требуются каждому программисту, но общение с ними редко приносит полное удовольствие.

Майк Болл

На рис. 5.3 представлены основные информационные связи и потоки компилятора. Исходная программа преобразуется компонентами компилятора в ряд промежуточных представлений и, в конечном итоге, в программу на языке назначения. В течение этих преобразований компоненты работают с общими символьными таблицами и при необходимости обращаются к обработчику ошибок.

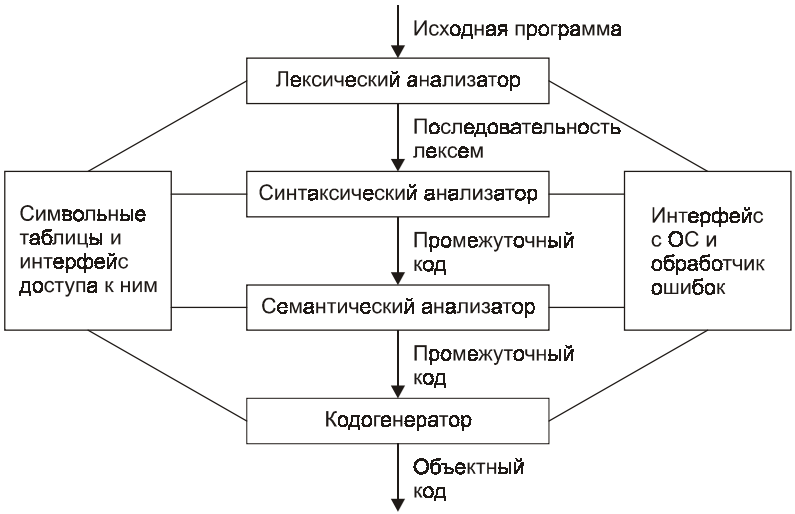


Рис. 5.3. Основные информационные связи и потоки компилятора

Далее мы расскажем об основных программах, входящих в состав компилятора. В сети Интернет подробный обзор таких программ можно найти на сайте Compiler Construction Tools (**http://catalog.compilertools.net**).

- *Драйвер (диспетчер, монитор) компилятора* — это программа, последовательно вызывающая все остальные компоненты компилятора.
- *Препроцессор* — программа, которая выполняет модификацию данных с целью их подготовки для ввода в другую программу. Модификация может заключаться в простом переформатировании или применении макрорасширений.
- *Анализатор* — компонент компилятора, осуществляющий последовательно:
  - *лексический анализ*, на котором входная программа, представляющая собой поток знаков, разбивается на лексемы — конструкции (символы и слова) языка;
  - *синтаксический анализ*, в процессе которого происходит разбор структуры программы;
  - *контекстный анализ*, во время которого выявляются зависимости между частями программы, включая анализ типов, областей видимости, соответствие параметров и т. п.
- *Оптимизатор* — компонент, выполняющий преобразования программы (как правило, на основе промежуточного представления) с целью улучшения ее характеристик (оптимизации) по одному или ряду параметров. Как правило, оптимизацию проводят по времени исполнения с целью получения более эффективной программы или по пространству — месту,

которое файл с откомпилированной программой будет занимать. Обратим внимание на то, что оптимизация улучшает некоторую характеристику программы ценой ухудшения других. На рис. 5.4 представлено три основных подхода к оптимизации.

- *Кодогенератор* — компонент, завершающий компиляцию и порождающий на основании промежуточного представления программы либо ассемблер, либо целевой (объектный) код.

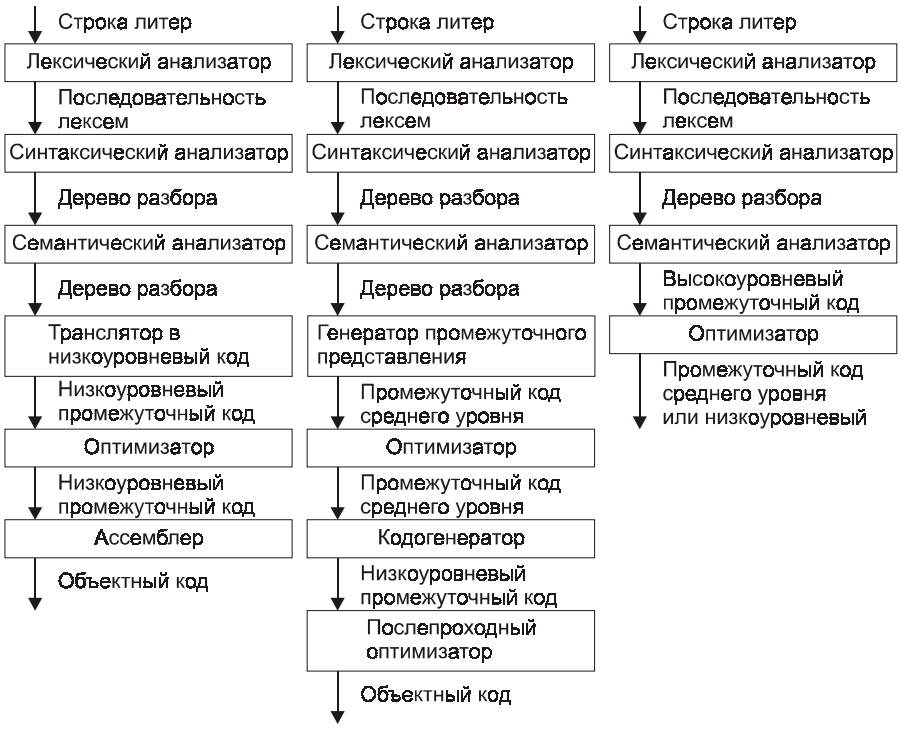


Рис. 5.4. Три подхода к оптимизации

5.2.4.3. Системы генерации (построения) трансляторов

Владею русским со словарем, французским, хинди, испанским, банту и другими с переводчиком.

Владимир Колечицкий

Системы генерации трансляторов начали появляться в середине 1960-х годов сразу после первых попыток реализации компиляторов. Рассмотрим системы построения трансляторов, причем сначала системы, основанные на LL-анализе и рекурсивном спуске, а затем LR-системы (*см. разд. 4.6*).

Одной из самых ранних разработок является система Brooker-Morris Compiler Compiler Тони Брукера (Tony Brooker) и Деррика Морриса (Derrick Morris) 1962 года, предназначенная для языка ATLAS. Здесь использовались расширения LL(1)-грамматик, по которым генерировалась процедура анализа методом рекурсивного спуска.

Еще одна система генерации трансляторов META была разработана Вэлом Шорром (Val Schorre) в 1964 году. По спецификации трансляции генерировался алгоритм на основе рекурсивного спуска, переводивший цепочку на входном языке в программу на языке низкого уровня. За данной системой последовала в 1965 году система TMG Боба Макклюра (Bob McClure), также предназначенная для анализа LL(1)-языков, но результатом трансляции был уже код на промежуточном языке.

В 1969 году под руководством Корнелиса Костера (Cornelis Koster) была разработана система CDL. В ней использовались аффиксные грамматики, а синтаксический анализ выполнялся методом рекурсивного спуска. Данная система развивалась дальше: в 1991 году появилась CDL3, имевшая расширенную функциональность. Семантические действия в этой системе реализуются на языке логического программирования.

Среди современных систем, предназначенных для работы с LL-грамматиками и их расширениями, можно отметить следующие:

- JavaCC фирмы Metamata. Для анализа используется комбинация рекурсивного спуска с перебором с возвратом (backtracking). Размер правого контекста (lookahead) можно задавать в спецификации;
- DEPOT4 — авторская разработка Юргена Лампе (Jurgen Lampe) (**http://www.math.tu-dresden.de/wir/depot4/**). Это система, генерирующая синтаксические анализаторы, работающие по принципу рекурсивного спуска. При этом допускается возможность регулированного отката при обнаружении неоднозначностей. Таким образом, при наличии требования на отсутствие левой рекурсии в грамматике сама грамматика может быть и не LL(1);
- LLGEN — часть проекта Amsterdam Compiler Kit (ACK) (**http://www.cs.vu.nl/vakgroepen/cs/ack.html**). Сгенерированные трансляторы используют метод рекурсивного спуска.

Обратимся теперь к системам "перенос-свертка" и LR-системам. На рубеже 1960-70-х годов Уильямом Маккиманом (William McKeeman), Джеймсом Хорнингом (James Horning) и Дэвидом Уортманом (David Wortman) была разработана система XPL. В ее основе лежит синтаксический анализ на основе расширения метода предшествования.

Более общий метод — LALR(1) анализ был положен в основу разработанной в 1972 году Стивеном Джонсоном (Stephen Johnson) системы yacc. Семантические действия реализуются на языке C. Система yacc широко использовалась

при разработке приложений системы Unix и стала очень популярна. Вскоре появились варианты yacc, расширявшие ее инструментарий и возможности. Можно выделить следующие разновидности системы:

- BISON — реализация yacc в рамках проекта GNU (**http://www.gnu.org/software/bison/bison.html**);
- PCYACC — реализация компании Abraxas Software (**http://www.abxsoft.com/pcyacc.htm**), включающая в себя различные утилиты, облегчающие процесс разработки синтаксических анализаторов;
- YACC++ — реализация компании Compiler Resources, Inc. (**http://world.std.com/~compres**). Это расширение, построенное на идеях объектно-ориентированного программирования. На выходе этой системы пользователь получает иерархию классов, реализующих трансляцию.

Огромная популярность yacc привела к тому, что многие последовавшие за ней системы имеют очень похожий синтаксис входных описаний. Однако далеко не все. Например, система GENTLE во многом опирается на CDL. Синтаксический анализ осуществляется LALR(1) автоматом (в самой системе для генерации этого автомата используется yacc). Семантики же реализуются на языке логического программирования, основанного на Prolog.

Целый ряд отечественных систем появился в 1970-1980-е годы. Отметим некоторые из них [Вооглайд, Серебряков 1989]. Система "Супер", разработанная в ВЦ АН СССР, использует LL(1)-грамматики и L-атрибутные вычислители. Система позволяет строить многопроходные трансляции, генерация кода происходит в промежуточный язык Лидер — ЛИнеаризованное ДЕРево.

В системе построения трансляторов ELMA (Таллиннский политехнический институт) используются грамматики предшествования. При этом делается попытка из грамматики произвольного вида с регулярными выражениями получить грамматику предшествования. Для реализации семантики используются атрибуты. В системе применяется двухэтапная схема генерации кода, причем в качестве промежуточного языка выступает Forth. LR(1)-грамматики используются в системе Сагет (Гомельский государственный университет).

Оригинальный подход используется в системе SYNTAX Бориса Константиновича Мартыненко [Мартыненко 1997]. Для анализа используется метод сверху вниз, однако, в отличие от многих аналогов, класс входных грамматик не ограничивается LL(1). На данный момент семантические вычисления производятся с помощью внешних процедур без использования атрибутов.

Дополним представленную выше картину указанием на экспериментальные системы, реализующие нестандартные подходы к проблеме спецификации и реализации трансляций. Одна из таких систем реализована в проекте ALE (Attribute-Logic Engine) — авторская разработка Боба Карпентера

(Bob Carpenter) и Джеральда Пенна (Gerald Penn) (**http://www.cs.toronto.edu/~gpenn/ale.html**). Система использует идеи из области искусственного интеллекта.

Заканчивая обзор существующих инструментов, определим, следуя [Вооглайд, Серебряков 1989], основные черты современной системы построения трансляторов.

- Автоматическое генерирование синтаксических процессоров по описанию — процесс, позволяющий использовать системы в системах автоматизирования сборки программ.
- Документирование синтаксического процессора — автоматическое создание листингов, текстовых описаний сгенерированных процедур.
- Поддержка отладочного режима работы.
- Высокая производительность анализатора.
- Широкий класс допустимых грамматик.
- Поддержка контекстной чувствительности анализатора.
- Технологическая поддержка разработки синтаксического процессора — использование репозиториев синтаксических компонент, библиотеки стандартных решений.
- Интеграция с универсальным языком — на уровне интерфейса прикладного программиста, на уровне семантических действий.
- Интегрированная система обработки ошибок — позиционирование ошибок во входном потоке и восстановление после ошибок.

5.2.4.4. Системы анализа корректности программного кода

Системы такого типа очень близки синтаксическому анализатору и являются средствами статического анализа программ. Обычно они разрабатываются для языков со слабым контролем типов и выполняют гораздо большее количество тщательных проверок, чем стандартные анализаторы. Это выполнение максимального количества типовых проверок и проверка соответствия параметров. Системы анализа корректности программного кода обычно выдают большое количество дополнительных предупреждений, касающихся всех подозрительных мест в программе. Ряд инструментов анализирует метрики программ. Во многих компаниях проверка исходных текстов программных продуктов системами анализа корректности программного кода является обязательной. Приведем следующие примеры таких систем:

- lint (компании Sun Microsystems, Inc. (**http://www.sun.com**)) для языка программирования C;
- PC-lint и FlexeLint (компании Gimpel Software (**http://www.gimpel.com**)) анализируют программный код на языках C и C++.

5.2.4.5. Интерпретаторы

Не существует фактов, есть лишь их интерпретация.

Фридрих Ницше

Интерпретатор — программа, осуществляющая непосредственное исполнение текста исходной программы пошаговым образом. Интерпретатор одновременно и транслирует, и выполняет заданную программу. Существуют языки программирования, для которых интерпретация программ, написанных на них, предпочтительнее компиляции. Это языки программирования с конструкциями, позволяющими создавать новые подпрограммы или модифицировать существующие динамически, во время выполнения программы. Независимо от языка интерпретация может быть предпочтительнее компиляции на этапе отладки программ.

Примером традиционно интерпретируемого языка программирования является Lisp. Однако для Lisp существуют и успешно применяются и компиляторы.

Примером одной из важных разновидностей интерпретаторов является *виртуальная машина Java*, интерпретирующая специальный низкоуровневый двоичный код — Java байт-код. Метод исполнения кода, основанный на интерпретации, делает его в большой степени независимым от архитектуры ЭВМ, на которой он исполняется.

5.2.4.6. Декомпиляторы

Декомпилятор — программа, позволяющая по программе на языке низкого уровня (обычно это коды некоторой машины) получить программу на высокоуровневом языке, в некотором смысле ей эквивалентную. Под *эквивалентностью* чаще всего понимают эквивалентность внешних проявлений работы исходной программы и декомпилированной.

При решении разнообразных практических проблем в области программирования зачастую бывает полезно декомпилировать имеющиеся программы, исходные тексты которых недоступны. Эта задача является весьма сложной и пока что может быть решена лишь в частных случаях, хотя методы, применяемые для всех частных случаев, имеют много общего. Одной из важных составных частей задачи декомпиляции является анализ потоков управления. Как правило, эта часть осуществляется при помощи исследования графа управления, полученного на этапе предварительного анализа программы на низкоуровневом языке. Дуги этого графа соответствуют условным и безусловным переходам между базовыми блоками программы, и он не содержит никакой информации об управляющих конструкциях высокого уровня, из которых должна состоять декомпилированная программа. Восстановление этих структур и является основной задачей анализа потоков управления при декомпиляции. Разумеется, ее специфика определяется языком высокого уровня, на котором должна быть написана итоговая программа, т. к. разные

языки, даже традиционно считаемые близкими и относящимися к одному семейству, имеют разные наборы допустимых управляющих конструкций.

Процесс, который необходимо выполнить при решении этой задачи, часто называют *структуризацией графа управления*. При структуризации каждую вершину графа управления надо сделать частью какой-либо высокоуровневой управляющей конструкции. Эти конструкции следует выбирать таким образом, чтобы не изменился порядок переходов управления между базовыми блоками.

Теория анализа потоков управления появилась давно, т. к. анализ активно применялся в компиляторах практически с самого момента их появления. Однако анализ потоков управления при компиляции имеет свою специфику: он используется в основном для целей оптимизации. Большинство алгоритмов анализа потоков управления, которые ориентированы на декомпиляцию, ставят своей целью уменьшение количества операторов `goto` посредством введения новых логических переменных, дублирования кода (code replication) или использования конструкций высокого уровня, недоступных в большинстве широко используемых языков, таких как Pascal, C или C++.

В работах Кристины Цифуентес (Cristina Cifuentes) (**http://www.itee.uq.edu.au/~cristina**) представлены алгоритмы, использующие минимальный набор инструкций высокого уровня: циклы типов `while` и `do ... while`, `if`- и `case`-выражения. Управление выполнением объемлющих конструкций из вложенных не допускается. Причем эти алгоритмы используют оператор `goto` лишь тогда, когда граф управления не может быть структурирован никаким иным способом в пределах перечисленных конструкций.

Анализ практических достижений в области декомпиляции языка Java выполнил Дэйв Дайер (Dave Dyer) (**http://www.javaworld.com**). Он сравнил между собой три наиболее известных к середине 1997 года декомпилятора: DejaVu, Mocha и WingDis. Дайер предложил систему тестирования декомпиляторов, основанную на интересной классификации допускаемых ими ошибок. В соответствии с этой системой все ошибки делятся на шесть категорий. "Тяжесть" ошибки возрастает с номером категории. Ошибки, в результате которых декомпилированная программа перестает собираться, но которые могут быть легко исправлены (например, отсутствие явного приведения типов там, где оно с очевидностью должно быть), относятся к первой, самой легкой категории, тогда как генерация правильного, но нечитаемого текста на Java считается ошибкой третьей категории. Заметим, что некоторые ошибки, проявляясь у одних декомпиляторов, полностью отсутствуют у других. Это свидетельствует о жизнеспособности такой классификации — она позволяет определить области, в которых одни декомпиляторы имеют преимущество над другими.

В настоящее время наиболее известны следующие декомпиляторы:

- Mocha и Crema входят в состав программного продукта Jbuilder компании Borland Software Corporation (**http://www.borland.com**));
- Source Tee Java Decompiler (компании SourceTee Software Co. (**http://www.srctec.com/decompiler/**));
- Jad (авторская разработка Павла Кузнецова (**http://www.sai.msu.su/sal/F/1/JAD.html**)).

5.2.4.7. Усложнители декомпиляции (шифраторы, обфускаторы)

Практические успехи в области декомпиляции языков программирования (и особенно языка Java) оказались столь велики, что в последнее время были проведены значительные исследования в области защиты программ от декомпиляции (**http://www.cs.washington.edu/homes/douglas/publish/**). Существует специальный класс программ (шифраторов, обфускаторов), усложняющих декомпиляцию. Такие программы используют два основных подхода.

- Общее (универсальное) усложнение.
- Целенаправленное усложнение (против конкретного декомпилятора).

Основная идея усложнителей заключается в ухудшении кода. Усложненный код после декомпиляции должен быть непонятен и нечитаем программистом.

5.2.4.8. Инструменты рефакторинга

Рефакторинг — процесс улучшения внутренней структуры текста программы, при котором не меняется ее внешнее поведение. Улучшение текста происходит уже после его написания. С помощью рефакторинга плохой или даже хаотический текст может быть переделан в хорошо спроектированный. Методов рефакторинга довольно много, в работе [Фаулер 2003] их описано около ста.

Рассмотрим простейший пример рефакторинга, называемого "выделение метода". Если есть фрагмент кода, который можно сгруппировать, то его следует преобразовать в метод, название которого объясняло бы его назначение. Это метод можно использовать практически всегда, когда код кажется слишком длинным или требующим комментариев, объясняющих его назначение.

Инструментов, поддерживающих рефакторинг, не так много. Основное требование, которое можно предъявить к таким инструментам, — это их надежность, дающая возможность проводить рефакторинг без обязательного повторного тестирования программы. Средство IntelliJ Idea (**http://www.jetbrains.com/idea/index.html**) — это мощное средство рефакторинга программного кода, разработанное для языка Java. В нем используется идея работы по синтаксическому дереву.

5.2.4.9. Системы управления компиляцией и построением программ

Системы управления компиляцией и построением программ в той или иной форме присутствуют во многих инструментальных системах, а также могут использоваться как явные утилиты, работающие с файлами проекта. Фактически эти системы являются генераторами команд, т. е., используя файл описаний, они создают последовательность команд для обработки командным интерпретатором. Приведем примеры таких систем:

- make (компании Sun Microsystems, Inc. (**http://www.sun.com**)), входящая в состав операционной системы Solaris;
- GNU Make (организации Free Software Foundations (**http://www.gnu.org/software/make/make.html**));
- Jmk — Make in Java (общинная разработка (**http://sourceforge.net/projects/jmk**)).

Основной алгоритм перечисленных систем (далее будем называть их make) таков. На основе совокупности инструкций, согласно которым должна производиться сборка и построение проекта, они проверяют временные метки и, при необходимости, перекомпилируют файл. В результате существенно экономится время сборки проекта.

Файл с инструкциями имеет свой синтаксис.

- Строка комментариев начинается с символа диез '#'.
- Пустые строки игнорируются.
- Любая строка, оканчивающаяся символом обратная наклонная черта '\', продолжается на следующую строку. Все символы пробела и табуляции вокруг наклонной черты сжимаются в один символ пробела.
- Если строка выглядит следующим образом `<цель>: [<зависимости>]`, то это означает, что цель определяется зависимостями. И, соответственно, для того чтобы добиться выполнения целей, необходимо выполнить зависимости.
- Строка, идущая за строкой с указанием цели, должна иметь вид `<пробел_или_табуляция> <команда>`. Указанная команда будет передана командному интерпретатору в случае, если цель нуждается в обновлении.
- Команда `[<пробел_или_табуляция>] <имя> = <значение>` позволяет изменять значение переменной. Любая строка, содержащая знак равенства, является макроопределением. Такая строка связывает строку `<значение>` с именем `<имя>`.
- Любое слово в файле описаний, начинающееся с символа '$', является использованием макроопределения. Включенное в любом месте файла

описание $(<имя>) будет заменено на его существующее макроопределение. Если <имя> является одиночным символом, то разрешено использование макроопределения в упрощенном виде $<имя>.

- Некоторые макроопределения являются предопределенными. Они могут использоваться только в правой части строки правила, в других макроопределениях и в строках команд. Приведем примеры предопределенных макроопределений:
  - $@ — полное имя текущей цели;
  - $* — имя текущей цели с отброшенным типом файла (суффиксом);
  - $? — список зависимостей, которые обновились с момента предыдущего получения цели;
  - $< — полное имя исходного файла, к которому применяется правило трансформации.
- Обычно существует некоторая база предопределенных правил суффиксов и динамических макроопределений. Они являются правилами по умолчанию для стандартного получения результирующих файлов по исходным файлам. Все они сведены в специальный файл. Например, правило трансформации .cc.o определяется как $(COMPILE.cc) $(OUTPUT_OPTION) $<. Это означает, что для того, чтобы получить файл .o (если есть файл .cc), надо выполнить указанную команду. Динамическое макроопределение $(COMPILE.cc) определяется в той же базе как $(CCC) $(CCFLAGS) $(CPPFLAGS) -c.

Пример файла с инструкциями приведен в листинге 5.1. Именование каталогов дано в стиле операционной системы Unix.

Листинг 5.1. Пример файла с инструкциями для утилиты make

```
BASE_DIR = /usr
BIN_DIR = $(BASE_DIR)/bin
LIB_DIR = $(BASE_DIR)/lib
# В результате такого описания получится, что значение
# переменной BIN_DIR стало равным /usr/bin, а LIB_DIR — /usr/lib.
OBJS = main.o classa.o classb.o
LIBS = libCstd.a libCrun.a
main: clean $(OBJS)
      $(CCC) -o $@ $(OBJS) -L $(LIB_DIR) $(LIBS)
# Разыменование переменной CCC, которая у нас не присутствует,
# но она может присутствовать в окружении.
# Использование $@ после параметра -o приводит к тому,
# что результирующий файл будет называться тоже main.
```

```
clean:
    rm -f *.o main core
# Удаление всех объектных файлов и core-файла, который может быть
# создан в результате некорректного завершения предыдущей компиляции.
classa.o: classa.h
# Указывает зависимость, т. е. когда будет изменен файл classa.h,
# файл classa.o потребует перекомпиляции.
```

При старте make может получить в качестве параметров имя файла определений и имя главной цели его работы. Если имя файла определений не указано, то в текущем каталоге ищется файл с именем makefile, а если он не находится, то make прекращает работу с сообщением об ошибке. Если не указано имя главной цели, то make по умолчанию берет в качестве главной цели первую цель, найденную ею в файле определений.

Существуют расширения стандартной системы make. Например, одно из них оценивает файл описаний с точки зрения его параллельной обработки и параллельного исполнения команд, необходимых для обновления главной цели. Команды могут быть выполнены параллельно только в случае, если они требуются для обновления абсолютно независимых целей. Иначе они будут по-прежнему исполняться последовательно.

5.2.5. Тестирование и отладка

Заблуждения, заключающие в себе некоторую
долю правды, самые опасные.
Адам Смит

Мы будем изучать вместе инструментарий, применяемый как на этапе тестирования, так и на этапе отладки. Очень часто оказывается, что один и тот же инструмент может указать на факт наличия ошибки, одновременно локализовать место ее возникновения и указать причину. Большая часть инструментов связана с технической стороной вопроса (**http://www.softwareqatest.com/qatweb1.html**).

5.2.5.1. Тестовые мониторы

Тестовый монитор состоит из трех основных компонентов.

- Ядро системы, содержащее основные программы тестирования и оформления результатов.
- Собственно тестовая база, включающая исходный тест и эталонные результаты пропуска тестов.
- Тестовое пространство, обычно содержащее специфический настроечный файл.

В результате запуска тестового монитора в тестовом пространстве начинают исполняться заданные сюиты из тестовой базы. На основе результатов тестирования можно сделать выводы об изменении качества программного продукта.

5.2.5.2. Средства отслеживания тестового покрытия

Системы такого типа предназначены для выявления тестового покрытия программы, в том числе участков кода, пропущенных при тестировании.

Важную роль здесь играет понятие линейного участка — *фрагмента* программы, на протяжении которого нет передачи управления. Если выполняется первый оператор такого участка, то будут выполнены и все остальные. Часто данные системы используют в качестве вспомогательного средства профайлер *(см. разд. 5.2.5.3)*.

Примерами систем являются:

- Rational PureCoverage (компании Rational Software Corporation (**http://www-306.ibm.com/software/rational/**));
- Java Test Coverage and Instrumentation Toolkits (компании Glen McCluskey & Associates LLC (**http://www.glenmccl.com**)).

5.2.5.3. Средства динамического построения профиля программы

Построение *профиля* программы позволяет обнаружить фрагменты кода, исполняемые при запуске программы. Полученная информация позволяет выявить в программе те места, которые надо оптимизировать. Работа со средствами динамического построения профиля программы включает три этапа.

- Компиляция и сборка программы с включенным режимом добавления кода с целью генерации информации для построения профиля.
- Запуск программы с целью получения информации, необходимой для построения профиля.
- Запуск профайлера для анализа полученной информации.

Результатом работы профайлера будет исходный текст программы, где для каждого линейного участка указана цифра: сколько раз этот исходный участок был исполнен, а также некоторая дополнительная статистика. Простейший пример результата работы профайлера приведен в листинге 5.2. В случае реальных программ, наиболее часто исполняемые участки надо максимально оптимизировать с целью ускорения работы всей программы. Примеры профайлеров следующие:

- prof (компании Sun Microsystems, Inc. (**http://www.sun.com**));
- gnuprof (организации Free Software Foundations (**http://www.gnu.org**));
- ProDelphi (авторская разработка Гельмута Адольфа (Helmuth Adolph) (**http://www.prodelphi.de**)).

Листинг 5.2. Пример результата работы профайлера

```
        #include <stdio.h>
        int counter(int i) {
500 ->      return(i);
        }
        int main() {
        int i;
1 ->    int s = 0;
        for (i=0; i<500; i++)
500 ->      s = s + counter(i);
1 ->    printf("%d \n", s);
        return 0;
        }
        Top 10 Blocks
        Line      Count
         3        500
         9        500
         7          1
        10          1
     4   Basic blocks in this file
     4   Basic blocks executed
 100.00  Percent of the file executed
  1002   Total basic block executions
 250.50  Average executions per basic block
```

5.2.5.4. Системы построения срезов программы

Программный срез состоит из всех операторов программы, которые могут влиять на значение некоторой переменной в некоторой позиции. Срезы были введены для использования:

- ❒ в отладке программ;
- ❒ в тестировании программ;
- ❒ в понимании смысла программ.

Причины возникновения данного направления очевидны. При работе с исходным текстом программы часто необходимо проследить формирование значения некоторой переменной. Это особенно актуально при отладке и сопровождении программ. В этом случае получение среза приносит существенное облегчение работы, поскольку резко снижается объем исследуемого кода.

Предположим, что во время тестирования мы установили, что некоторая переменная в некоторой позиции содержит неверное значение. Получив соответствующий срез, мы локализуем проблему. В дополнение к сказанному, срезы позволяют разбить большую программу на небольшие компоненты. Это способствует более простому пониманию смысла программы, что очень актуально при сопровождении программных продуктов. Впоследствии часть этих компонентов может быть использована при создании новой программы. В статье [Badger, Weiser 1988] продемонстрировано, как срезы могут быть использованы для автоматической параллелизации. Однако основной областью применения срезов остается отладка и сопровождение программ.

В начальном понимании понятие срез (статический срез) учитывает все возможные пути исполнения программы. Однако обычно на практике мы имеем дело с частным случаем, в котором программа работает некорректно, и, естественно, нас интересует причина происходящего. Таким образом, вместо того чтобы отслеживать все множество входных данных, мы будем рассматривать срез, который сохранит свойства программы для определенного входа. Такой тип среза был обозначен термином *динамический срез*.

Большинство существующих методов вычисления динамического среза основано на так называемом *обратном анализе*, когда после исполнения программы записывается ее трассировка, и затем алгоритм отслеживает эту трассировку в обратном направлении для получения зависимостей по данным и по контролю, которые в свою очередь используются для вычисления динамического среза. Существуют методы, вычисляющие динамический срез в процессе исполнения программы и не записывающие трассировку.

Практически все существующие методы вычисления динамического среза рассчитаны на работу с программами, написанными в стиле структурного программирования. Лишь немногие работы, например [Ball, Horwitz 1993], предлагают подход для вычисления статических срезов для неструктурированных программ, модифицировав понятие зависимости по контролю для данного случая.

В существующих алгоритмах основной целью вычисления динамического среза является определение той части кода, которая влияет на значение интересующей переменной, путем выявления зависимостей по данным и по контролю *(см. разд. 4.3.3.6)* в трассировке. При определении динамического среза важно найти не только части, влияющие на процесс вычисления интересующей переменной, но также и части, не влияющие на него. Чем больше таких не влияющих частей будет найдено, тем меньший динамический срез может быть получен.

5.2.5.5. Отладчики

Отладчик — это программа, помогающая анализировать поведение отлаживаемой программы, обеспечивая ее трассировку. При этом большинство отладчиков позволяют выполнять остановки в указанных точках или при заданных условиях, просматривать текущие значения переменных, ячеек памяти, регистров процессора и при необходимости изменять эти значения.

Нужные для отладчика данные порождает компилятор при включенном режиме генерации отладочной информации. Это данные о типах, начале и конце блоков, специфических конструкциях и т. п.

Кроме классических отладчиков существуют еще несколько групп инструментов.

- *Средства динамической отладки распределения памяти* — специальный инструментарий, позволяющий анализировать распределение памяти в процессе исполнения программы.
- *Средства отладки многопоточных и параллельных приложений* — приложения, которые в некоторой точке могут разветвиться для выполнения параллельной работы, потом попадают в точку ожидания и опять выполняются последовательно. Примеры — средства компании Sun Microsystems, Inc. (**http://www.sun.com**):
  - thd (Thread Analiser) — нитевой анализатор, который исполняется для того, чтобы отслеживать многопоточные приложения;
  - locklint — анализатор корректности взаимных исключений;
  - looptool — анализатор распараллеливания программ.

5.2.5.6. Системы отслеживания проблем (ошибок)

Системы отслеживания проблем обычно представляют собой интерфейс к обыкновенной базе данных. Наиболее сложный вопрос при разработке таких систем — отбор информации, которая связана с проблемой (ошибкой). Вот одна из возможных структур такой информации:

- основная информация об ошибке:
  - номер ошибки;
  - краткое описание проблемы;
  - ключевые слова;
  - важность ошибки (приводит к критическим последствиям или создает незначительные неудобства);
  - приоритет (сумеем ли исправить в этой версии);
  - состояние *(см. разд. 3.2.5.2)*;
- место проявления ошибки:
  - программный продукт;
  - версия продукта;
  - компонент;
  - особенность;
- персоналии:
  - ответственный менеджер;
  - ответственный инженер;

- инженер, выставивший ошибку;
- адреса рассылки;

❒ анализ ошибки и исправление:

- подробное описание ошибки;
- тест, на котором проблема проявляется;
- предлагаемое исправление;
- комментарии;
- информация для пользователей;

❒ влияние исправления:

- интеграция регрессионного теста в тестовую базу данных;
- влияние исправления на документацию к программному продукту;
- влияние исправления на другие компоненты.

Примерами систем отслеживания проблем являются:

❒ Bugzilla, применяемая для сбора информации об ошибках гипертекстового браузера Mozilla (**http://bugzilla.mozilla.org**);

❒ Open Source Web Browser (авторская разработка Адама Сигеля (Adam Siegel) (**http://sourceforge.net/projects/btrack**)).

5.2.6. Ввод в действие

Основная группа систем данного процесса — это *системы инсталляции*. Укажем самые популярные инсталляторы для операционной системы Windows (**http://www.softshape.com/swrus/faq_swrus.html**):

❒ MindVision's Installer VISE — мощнейший профессиональный инсталлятор, используемый даже крупными корпорациями;

❒ WISE — инсталлятор, известный самым маленьким размером получающегося дистрибутива. Имеет богатый язык скриптов;

❒ InstallShield Professional/Express — старейший продукт, обладающий всеми необходимыми возможностями для создания сколь угодно сложных сценариев установки;

❒ CreateInstall — простой, легкий в освоении и компактный инсталлятор.

5.2.7. Сопровождение

Можно выделить следующие основные группы инструментов, используемых при сопровождении программ:

❒ связанные с технической стороной сопровождения (трансляторы, отладчики, системы управления версиями текстов и т. п.). Поскольку сопровождение представляет собой мини-модель жизненного цикла, то инструменты,

применяемые во всех процессах жизненного цикла, могут понадобиться и здесь;

- связанные с анализом и пониманием кода. Это специальные средства реинжиниринга и обратной инженерии. Также сюда мы относим средства восстановления спецификаций, конверторы и другие системы;
- связанные с организационной стороной сопровождения. Эти средства дают возможность выполнения административных функций (например, ведения базы данных ошибок) сопровождающему программный продукт.

Наиболее интересными здесь являются инструменты второй группы, примеры которых приведем ниже.

- Средство IntelliJ Renamer (**http://www.jetbrains.com/idea/index.html**) — это мощное средство для преобразования программ и поиска в программном коде, разработанное для языка Java. Оно позволяет производить переименование методов, пакетов, полей, переменных и параметров методов с автоматическим исправлением вхождений. IntelliJ CodeSearch позволяет делать поиск в исходном коде на языке Java, например, вызовов определенных методов или определения классов, реализующих некоторый интерфейс. Это средство корректно работает с перегруженными методами различных классов с одинаковыми именами. В этих средствах используется идея работы по синтаксическому дереву.
- Система RescueWare (**http://www.relativity.com/products.htm**) является системой производства программ с новой бизнес-логикой, преобразуя и используя части существующих программных ресурсов. Для осуществления этих операций система предлагает разделение задачи на несколько подзадач. Сначала проходит стадия понимания программ [Бульонков, Бабурин 2000]. На этом этапе выделяются такие сущности, как программный уровень, потоки данных, карта вызовов. Далее проходит стадия изоляции определенной бизнес-логики. На этом этапе выделяются компоненты, пригодные для переиспользования. Впоследствии созданные компоненты используются для генерации новой программной системы с учетом особенностей новых платформ. В таких подходах ключевым моментом зачастую является преобразование программы на процедурном языке в программу на объектно-ориентированном языке. Для осуществления такого преобразования объекты должны быть идентифицированы из процедурных частей. Объектно-ориентированные программы (в сравнении с процедурно-базируемыми) считаются более простыми в сопровождении.

Существуют также системы, которые предназначены для анализа исходных кодов с целью получения высокоуровневых абстракций. Так, например, существуют средства [Niere, Wadsack, Zundorf 2000], которые анализируют код на языке Java с целью генерации диаграмм UML. Данный подход базируется

на использовании сетей рассуждений, построенных на нечеткой логике [Jahnke, Heitbreder 1998]. Также существуют средства, которые предоставляют среду разработки для объектно-ориентированных языков, используя диаграммы UML. Они позволяют не только генерировать код по диаграммам, но и анализировать его с целью построения такой диаграммы. Это позволяет более наглядно представлять архитектуру разрабатываемой системы и улучшать ее понимание.

5.2.8. Завершение эксплуатации

Основная группа систем данного процесса — это *системы деинсталляции*. Такие системы выполняют действие, обратное действию инсталляции, и, как правило, входят в инсталляционный пакет.

5.3. Универсальный инструментарий

Напомним, что в этом разделе под универсальностью инструментария будем понимать возможность его применения в нескольких процессах.

5.3.1. Инструменты работы с текстом

Существующие на сегодняшний момент системы можно подразделить на несколько классов. Так, некоторые из них используются как простые средства для обработки текстов, другие же представляют собой мощные автоматизированные программные комплексы. В этом разделе перечислены наиболее значимые из них.

5.3.1.1. Средства, базирующиеся на регулярных выражениях

Развитыми средствами определения шаблонов и осуществления поиска являются средства семейства grep в операционной системе Unix. Как grep, так и egrep могут искать участки текста, используя регулярные выражения. Наряду со специальными утилитами используются механизмы, встроенные в некоторые языки обработки текстов, например Perl (**http://www.perl.org**).

Эти средства используют текстовый поиск и в них не заложена семантика входного текста. А следовательно с их помощью нельзя отличить поиск по регулярным выражениям в произвольном тексте от поиска в исходном тексте программ. Можно провести аналогии с использованием архивирующих систем: те из них, которые предназначены для архивации текста, будут обладать гораздо меньшей эффективностью для архивации потоков звука и видео, и наоборот. Поэтому такие средства обладают рядом ограничений.

- Написание конкретного шаблона для поиска в исходном тексте может быть затруднительным, а иногда и невозможным. Например, пусть требуется

локализовать два последовательных оператора цикла: цикл типа `for`, следующий за циклом типа `while`, причем оба с произвольными условиями и блоками операторов, на одном и том же уровне вложенности. Возможная спецификация для grep может быть такой: `.*while.*for`. Однако такая спецификация grep отыщет не только последовательные операторы цикла, но и вложенные. Дополнительно будут найдены такие последовательные операторы, которые непосредственно не следуют друг за другом. Кроме того, оба требуемых ключевых слова могут оказаться в комментарии.

- Вторая проблема, связанная с использованием grep, заключается в написании шаблонов для отыскания деклараций. Рассмотрим написание шаблона для поиска декларации целой переменной `x` и символьной `c`. Поскольку порядок деклараций на одном и том же уровне вложенности в большинстве случаев не столь важен, они могут оказаться в любом порядке в исходном тексте. Поэтому для отыскания, например, двух деклараций с помощью регулярных выражений надо применять конструкцию такого вида: `(.*int[ ]*x[ ]*;.*char[ ]*c[ ]*;) | (.*char[ ]*c[ ]*;.*int[ ]*x[ ]*;)`. Сложность же такой спецификации для более чем двух отыскиваемых деклараций очевидна. Если в исходном тексте была декларирована инициализированная целая переменная (`int x=0;`), то поиск для нее окажется неудачным в любом случае.
- Третья проблема, которая может возникнуть при использовании средств семейства grep, связана со специальными символами. Обычно эти средства воспринимают символ "новая строка" как разделитель даже тогда, когда мы ищем выражение в многострочном контексте. Это особенно мешает поиску в исходном тексте, поскольку зачастую предмет поиска охватывает несколько строк. К тому же необходимость постоянно учитывать разделители (пробелы, символы "новая строка", символы табуляции) затрудняет составление регулярных выражений.

Регулярные выражения для поиска по шаблонам используются в утилитах ed, sed, awk и perl:

- ed и sed являются текстовыми преобразователями. Они позволяют находить и заменять фрагменты, базируясь на регулярных подвыражениях (в случае sed — многострочных). Они тоже воспринимают входной текст как поток символов, что некорректно;
- awk воспринимает входной текст как набор записей, и, несмотря на то, что это уже более подходящая абстракция, нежели символ, она все же не адекватна в применении к программному тексту. Семантически разные части могут оказаться в одной записи. Например, `for(i=0;i<5;i++){` — это все одна запись;
- perl, в частности версии 5, обладает более широким механизмом регулярных выражений, но даже и они остаются не столь удобными для поиска в исходном тексте программы, поскольку не учитывают его особенности.

5.3.1.2. Средства поиска различий

Средства поиска различий предназначены для обнаружения разницы между файлами. Такая необходимость возникает достаточно часто. Например, для сравнения старой и новой версии одного файла или сравнения изменений, внесенными двумя разными людьми на основе одной исходной версии. Стандартным результатом работы таких средств являются строки файлов, которые нужно изменить для того, чтобы привести файлы в соответствие друг с другом. В большинстве случаев находится минимальный достаточный набор различий.

Примеры таких систем следующие:

- diff (компании Sun Microsystems, Inc. (**http://www.sun.com**)) входит в состав операционной системы Solaris;
- GNU diff (организации Free Software Foundations (**http://www.gnu.org/directory/diffutils.html**)) представляет собой совокупность нескольких утилит:
  - diff — основная утилита для нахождения различий между двумя файлами;
  - cmp — утилита, выдающая информацию о номерах строк, в которых в файлах есть различия;
  - diff3 — утилита, находящая различия между тремя файлами;
  - sdiff — утилита для интерактивного объединения двух файлов в один.

5.3.1.3. Средства поиска на основе шаблонов

Средства поиска на основе шаблонов в качестве входных данных получают исходный код на языке программирования и запрос на языке шаблонов (образцов). На выходе выдается информация о том, какие строки исходного программного кода соответствуют спецификации шаблона.

В запросе можно также указать поиск последовательности операторов. В таком случае запрос понимается как поиск указанной последовательности операторов в любом месте исследуемого кода. Вся последовательность воспринимается как один запрос.

Для того чтобы специфицировать запрос к исходному коду, используется язык образцов. Язык образцов для спецификации запроса является расширением базового языка. Расширения включают набор специальных символов, которые используются для замены синтаксических сущностей базового языка. Без учета дополнительных расширений запрос может состоять из участка программы для осуществления конкретного поиска по исходному тексту. Когда же спецификация запроса включает дополнительный символ или символы, то они играют роль абстрактных шаблонов. Шаблоны потенциально могут быть заменены реальными фрагментами кода в соответствии с семантикой того или иного символа.

Специальные символы языка запросов применяются для замены синтаксических сущностей, таких как:

- декларация произвольной переменной с указанием произвольного типа;
- произвольный идентификатор, константа или выражение;
- произвольный формальный параметр или группа формальных параметров;
- произвольный оператор или группа операторов.

Запросы могут быть следующих типов:

- поиск декларации (как локальной, так и глобальной);
- поиск определения функции, спецификация кода которой описана в запросе;
- поиск группы операторов в произвольном участке кода.

Для примера приведем несколько конструкций запросов. В качестве базового языка выберем язык C.

- Найти все операторы цикла `while`, в которых происходит сравнение какого-нибудь выражения на неравенство его нулю. Запрос будет выглядеть так: `while (# != 0) @`. На основе данного запроса система производит поиск всех операторов `while`, удовлетворяющих вышесказанному условию, на любом уровне вложенности. Знак `#` — это шаблон для произвольного выражения, а `@` — шаблон для произвольного оператора. Такой запрос может оказаться полезным, если необходимо выявить все конструкции в программе с таким условием, с целью их замены на `while (#) @`.
- Найти все вхождения трех подряд идущих операторов `if`. Запрос будет выглядеть так: `if (#) @ if (#) @ if (#) @`. Целью такого запроса может стать локализация потенциальных кандидатов на преобразование последовательных операторов `if` в оператор `switch`.
- Найти все вхождения оператора `if`, где в качестве условия возможно ошибочное использование оператора присваивания `=` вместо равенства `==`. Запрос будет выглядеть так: `if (# = #) @`. Целью такого запроса может стать локализация часто встречающейся ошибки в коде на языке C, когда в выражениях сравнения используется оператор присваивания.
- Найти последовательности операторов, в которых три или более операторов `if` встречаются подряд, причем возможно их разделение другими операторами. Запрос будет выглядеть так: `if (#) @ @* if (#) @ @* if (#) @`. Целью такого поиска может стать локализация участков с большой цикломатической сложностью. В этом примере имеется восемь различных путей управления через данную последовательность операторов.

5.3.1.4. Обозреватели и базы данных программ

Обозреватели генерируют базу данных программы, куда в качестве сущностей включаются имена файлов, имена функций, переменные и другие сущности программы с хорошо определенными отношениями между ними. Чтобы получить информацию о том, какая функция вызывает определенную процедуру или глобальные данные, используемые какой-то определенной функцией, могут быть сделаны запросы к этой базе данных программы.

Хотя такие средства очень мощны и полезны, они все же поддерживают ограниченный диапазон запросов, которые базируются на концепции "сущность-отношение-атрибут". При таком рассмотрении отсутствует структурность, присущая исходному программному коду.

Приведем примеры инструментов, разработанных для выполнения запросов к исходному коду и упрощающих просмотр текста:

- Omega [Linton 1984];
- CIA [Chen, Nishimoto, Ramamoorthy 1990] и CIA++ [Grass 1992];
- Microscope [Ambras, O'Day 1987].

5.3.1.5. Средства обнаружения плагиата

Инструменты для определения плагиата в программах (особенно академических) в основном базируются на метриках Холстеда. Другая категория таких инструментов использует статическое дерево исполнения (граф потока управления) программы, чтобы составить "отпечаток пальцев" программы. Далее дерево исполнения анализируется, и на основе анализа делаются выводы о том, была ли программа заимствована. Основное ограничение таких систем состоит в том, что сравнение программ в них зависит от статистической информации.

В работах группы исследователей Бернского университета (**http://www.iam.unibe.ch/~scg/**) представлено средство обнаружения повторяющихся участков кода. В нем используется техника абстракции от языка. Это осуществляется с помощью подхода, названного авторами *визуализацией кода*. Он основан на первоначальном преобразовании кода к удобному для дальнейшего исследования виду. Данный подход базируется на преобразовании текста, а не на синтаксических сущностях.

5.3.1.6. Текстовые редакторы

Текстовый редактор — программа, позволяющая вводить в файл символьную информацию и модифицировать ее. Можно выделить несколько разновидностей текстовых редакторов.

- *Строковые* редакторы, позволяющие смотреть и редактировать текст только на одной строке. При этом текст воспринимается как последовательность строк, разделенных символом "конец строки".

- *Потоковые* редакторы, позволяющие воспринимать текст в виде потока символов. При этом признаки конца строки также являются символами.
- *Экранные* редакторы, позволяющие перемещать по тексту окно, в котором можно передвигать курсор и устанавливать его в нужные позиции. Многие экранные текстовые редакторы позволяют видеть на экране текст в том виде, в котором он будет напечатан (концепция What You See Is What You Get — WYSIWYG).

Сейчас распространены универсальные текстовые редакторы, которые избыточны для большинства конкретных задач. Рассмотрим несколько уровней текстовых редакторов, каждый из которых обладает дополнительными возможностями по сравнению с предыдущими:

1. Редакторы для составления программ. Их основные возможности — поддержка средств работы с блоками текста, поиск и замена по файлу.
2. Редакторы для подготовки статей без графики и сложных формул. Добавляются возможности форматирования абзацев, расстановки переносов и разбиения текста на страницы.
3. Редакторы для подготовки статей со сложными рисунками и формулами. Добавляются возможности включения графики в текст, поддержки многих кодировок и управления шрифтами.
4. Редакторы для подготовки книг и журналов. Добавляются возможности работы с большими файлами и сбор оглавления.
5. Редакторы для подготовки научно-технической документации. Добавляются возможности поддержки листов изменений, колонтитулов и предметных указателей.

Редакторы для составления программ могут учитывать синтаксическую структуру обрабатываемой программы и выполнять дополнительный анализ и проверку *(см. разд. 5.3.1.7)*.

Вот несколько наиболее известных и популярных текстовых редакторов:

- Microsoft Word (компании Microsoft (**http://www.microsoft.com**)) в операционной системе Windows и аналогичный ему по возможностям редактор StarOffice (компании Sun Microsystems, Inc. (**http://www.sun.com/software/star/staroffice/**)) в Unix;
- Биллом Джоем (Bill Joy) был разработан полноэкранный текстовый редактор для Unix — vi. Этот редактор до сих пор является одним из самых используемых стандартных текстовых редакторов на компьютерах с операционной системой Unix.

5.3.1.7. Синтаксически-ориентированные редакторы

Синтаксически-ориентированные редакторы учитывают при работе структуру обрабатываемого текста. Они позволяют реализовать следующие возможности:

- редактирование текста на уровне смысловых конструкций используемого языка;

- избавление пользователя от необходимости детально знать правила записи конструкций языка программирования;
- ускорение ввода программы за счет подстановки языковой конструкции целиком при нажатии одной функциональной клавиши;
- ведение непрерывного контроля правильности программы, включая синтаксис и полную статическую семантику;
- обеспечение автоматического форматирования текста программ и ее просмотра в удобном режиме.

Примеры синтаксически-ориентированных редакторов следующие:

- EMACS (организации Free Software Foundations (**http://www.gnu.org/software/emacs/**)) — редактор, в основе которого лежит LISP-система;
- MENTOR — исторически первая система структурного редактирования;
- ALOE — генератор структурных редакторов в рамках проекта GANDALF.

5.3.1.8. Гипертекстовые средства

Средства исследования и просмотра с гипертекстовыми возможностями позволяют просматривать и редактировать текст (обычно программный код), используя специальные средства браузеров, такие как выделение цветом, стилем, гипертекст и т. п. Разработаны и продолжают разрабатываться системы, целью которых является исследование и просмотр исходного текста крупных проектов с открытым исходным текстом. В качестве средства просмотра используется обычный HTML-браузер. Примерами таких средств являются:

- Linux Cross-Reference — LXR (**http://lxr.linux.no**) было создано как средство просмотра ядра операционной системы Linux;
- Mozilla Cross-Reference (**http://lxr.mozilla.org**) — средство просмотра исходных текстов браузера Mozilla;
- FreeBSD Unix Cross-Reference (**http://lxr.linux.no/**) — средство просмотра текстов операционной системы FreeBSD Unix.

5.3.2. Системы документирования

Существуют различные *генераторы документации*, большинство из которых основано на различных принципах сбора информации по исходным текстам программ. Вот примеры таких программ:

- javadoc (компании Sun Microsystems, Inc. (**http://www.sun.com**)) — генератор документации Java. Он считывает исходные тексты программ на языке Java, ищет комментарии специального вида и сопоставляет их с классами и методами, которые они описывают. Собрав все элементы,

генератор придает им вид HTML-структуры и сохраняет в виде файла. Комментарии для генератора похожи на блочные комментарии в языке C, но имеют две звездочки после первой наклонной черты. Существует несколько ярлыков — символов, имеющих специальное значение (например, знак `@`). Так, комбинация символа и слова `@author` заставит программу использовать специальные команды HTML, выделяющие имя автора в программе;

- ToolBook — инструментальная система разработки информационных приложений, создания гипермедиа-приложений (электронной документации, словарей, справочников и энциклопедий) и интерактивных обучающих программ. Отличительная особенность данной системы — использование метафоры книги в качестве основы создаваемых приложений. Приложение называется книгой, состоит из страниц и обладает всеми свойствами Windows-приложений [Смирнов, Керов, Дерюшев 1998];
- система CWEB Дональда Кнута предлагает метаязык, строящийся на основе двух формальных языков: языка программирования и языка разметки текста (форматирования). Первоначально Кнут использовал Pascal и TeX, соответственно, но одновременно заложил подход достаточно общий, чтобы впоследствии круг поддерживаемых языков программирования расширился (сегодня он включает C, C++, Java, Lisp, ML и др.). CWEB предполагает (**http://www.computerra.ru/offline/2001/387/7737/**), что все изменения, которые вносятся в программу, производятся на уровне CWEB-"метаисходника", который подвергается двоякой обработке (рис. 5.5):
  - программа CTANGLE генерирует из него файл на языке программирования C, подвергающийся затем трансляции (причем он снабжается директивами препроцессора, предназначенными для того, чтобы отладчик и другие инструменты направляли его к CWEB-"метаисходнику", а не к тексту на C);
  - программа CWEAVE создает TeX-файл, содержащий код секции, оформленный в соответствии с традиционными типографскими соглашениями о публикации исходных текстов, и пояснения к нему. Допускается также генерация в формат PDF с гиперссылками.

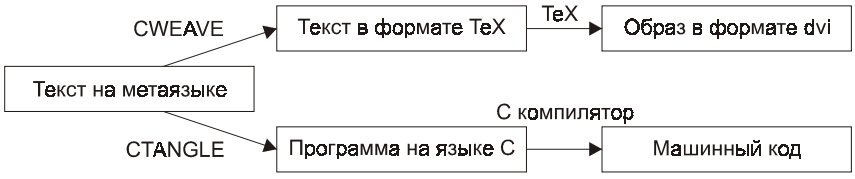


Рис. 5.5. Основные операции системы CWEB

5.3.3. Системы разработки интерфейсов

Системы разработки интерфейсов в начале 90-х годов прошлого века составляли большую долю в инструментарии. Сейчас такие системы входят составной частью в CASE-средства. Приведем примеры некоторых инструментов, которые можно использовать для разработки интерфейсов:

- C++ Visual Studio (компании Microsoft (**http://www.microsoft.com**));
- C++ WorkShop Visual (компании Sun Microsystems, Inc. (**http://www.sun.com**));
- Delphi Suite (компании Borland Software Corporation (**http://www.borland.com**));
- средства построения графического интерфейса в Java (компоненты и контейнеры).

5.3.4. Системы управления базами данных

База данных — это последовательная совокупность данных, отображающая состояние объектов и их отношения в рассматриваемой предметной области и хранящаяся с минимальной избыточностью.

Система управления базами данных (СУБД) — совокупность языковых и программных средств, предназначенных для создания и ведения базы данных. СУБД располагается между собственно физической базой данных и пользователями системы.

Основная функция СУБД — предоставление пользователю базы данных возможности работать с ней, не вникая в детали на уровне аппаратного обеспечения. Другие функции СУБД — реализация транзакций, обеспечение безопасности данных и т. п.

По типу поддержки модели данных различают несколько классов СУБД:

- реляционные СУБД;
- объектные СУБД;
- логические СУБД.

Значительно менее популярными в настоящее время являются СУБД, поддерживающие модели данных: иерархическую, сетевую, плоский файл, реляционную с вложенностью, семантическую и многомерную.

Заметим, что в большинстве случаев при проектировании базы данных сначала строится инфологическая модель предметной области, не зависящая от конкретной модели представления информации (и, следовательно, СУБД). Для ее создания обычно используются два источника информации:

- предметная область (в которой проводятся исследование, абстрагирование, поиск);

- информационные потребности пользователя (на основе которых выполняются описания).

Крис Дейт (Chris Date) [Дейт 2000] утверждает, что "и с экономической, и с теоретической точек зрения можно сказать, что реляционный подход наиболее важный (и это положение дел, по-видимому, не изменится в обозримом будущем)". Вот некоторые наиболее известные реляционные СУБД:

- Microsoft SQL Server (компании Microsoft (**http://www.microsoft.com**));
- DB2 (компании IBM (**http://www-306.ibm.com/software/data/**));
- Oracle (компании Oracle (**http://www.oracle.com**));
- Access (компании Microsoft (**http://www.microsoft.com**)).

Две следующие разработки относятся к классу расширяемых реляционных СУБД, позволяющих высококвалифицированным пользователям настроить систему заданным образом (например, определить собственные типы данных, структуры хранения, функции и т. п.).

- POSTGRES (создана в университете Беркли (**http://db.cs.berkeley.edu**)).
- Starburst (компании IBM (**http://www.research.ibm.com**)).

5.3.5. Системы управления базами знаний и экспертные системы

5.3.5.1. Системы искусственного интеллекта

Военно-промышленный комплекс сворачивает свои программы
по созданию оружия с элементами искусственного интеллекта —
опытный образец "умной" бомбы так и не удалось выпихнуть из самолета.

Анекдот о проблемах реализации систем искусственного интеллекта

Искусственный интеллект (artificial intelligence) — одна из областей информатики, направленная на моделирование и решение задач, связанных с обработкой символьной информации, логикой и естественным языком [Нильсон 1985]. Искусственный интеллект ставит перед собой и более серьезную задачу построения теории интеллекта, базирующуюся на обработке информации. Отдельные ветви этой области касаются психологии, философии и лингвистики. Базы знаний и экспертные системы — это два наиболее известных типа систем, связываемых с искусственным интеллектом.

К *системам искусственного интеллекта* можно предъявить различные требования (**http://asu.pstu.ac.ru/structure/prepod/solov.htm**).

- *Работа с неполной и неточной информацией.* Системы должны поступать "как крутой детектив". На основе неполной и неточной информации они разрабатывают первый вариант алгоритма, который потом совершенствуется на основе новой информации, получаемой системой.

- *Понимание плохо формализуемого.* Системы должны быть "как кулинарная книга". Например, "добавить по вкусу", "варить до готовности" и т. п.
- *Понимание смысла вопроса.* Например, на вопрос "Вы не скажете, который час?" в некоторых случаях вполне разумным будет ответ "Меня зовут Лена".
- *Наличие возможности обучения и объяснения.* Системы должны "напоминать толмача". Они должны иметь возможность обучения и объяснения. Например, будет полезнее, если система в области медицины на вопрос о наиболее эффективном лечении подробно объяснит, "что с такой болезнью давно пора помирать", вместо того чтобы прописывать цианистый калий.
- *Самообучение.* Системы должны не только хранить закономерности, но и показывать, как они используются при поиске решения. Например, при конструировании электрического стула система должна найти закон Ома и использовать его в расчетах.

5.3.5.2. Механизмы выводов на знаниях

Жизнь — это искусство делать верные выводы из неверных посылок.

С. Батлер

Под *механизмом вывода* в экспертных системах будем понимать ту часть экспертной системы, в которой содержатся общие знания о схеме управления решением задач. Машина вывода выполняет две основные функции [Гаврилова, Хорошевский 2001].

- Определение порядка просмотра и применения правил.
- Просмотр существующих фактов из рабочей памяти (базы данных) и правил из базы знаний и добавление в рабочую память новых фактов.

В зависимости от проблемной области (пространства поиска), будем классифицировать выводы следующим образом:

- вывод в одном пространстве;
- вывод в иерархических пространствах;
- вывод в альтернативных (параллельных) пространствах;
- вывод с использованием нескольких моделей;
- вывод в динамически изменяющейся проблемной области;
- вывод при неточных данных.

Вывод в динамической проблемной области и вывод при неточных данных характеризуются или присоединением к обычному выводу дополнительных функций, или своеобразной спецификой, уникальными (возможно нигде более не используемыми) методами.

Перечислим основные типы выводов на знаниях.

- *Логический вывод* — рассуждение с целью установления истинности какого-либо утверждения. Это рассуждение опирается при этом на аксиомы данной теории и ряд допущений или гипотез и состоит в последовательном переходе от этих отправных точек к новым, логически связанным с предыдущими предложениями. Аксиоматические теории представляются в виде формальных систем или формальных теорий, в которых все предложения, в том числе и аксиомы, записываются как формулы некоторого формализованного языка, а все способы рассуждения сведены к небольшому числу чисто формальных правил вывода.
- *Метод резолюций Дж. Робинсона* — это один из возможных путей оптимизации процесса поиска вывода. Рассмотрим метод резолюций в языке исчисления предикатов первого порядка — формальном языке, в котором можно выразить большое разнообразие утверждений.
  - Исходная логическая формула исчисления предикатов должна быть приведена к стандартной форме — множеству дизъюнктов. Следующие идеи используются для достижения этой цели. Формула может быть сведена к предваренной нормальной форме, в которой матрица не содержит кванторов, а префикс есть их последовательность. Не содержащая кванторов матрица может быть сведена к конъюнктивно нормальной форме. Сохраняя противоречивость формулы, можно избавиться от квантора существования (сколемизация). Таким образом, любая формула может быть представлена множеством дизъюнктов.
  - Основная идея метода резолюций — проверить, содержит ли `S` (множество дизъюнктов) пустой дизъюнкт `(s)`. Если да, то `S` невыполнимо. Если не содержит, то проверяется следующий факт: может ли быть `s` получен из `S` и т. д. Таким образом, метод резолюций можно рассматривать как специальное правило вывода для порождения из `S` новых дизъюнктов.

О практическом применении метода резолюций

Следует заметить, что метод резолюций используется в так называемых языках логического программирования. Например, на языке Prolog запись программы будет близка к записи в стандартной форме.

- *Обратный вывод С. Ю. Маслова* — это метод, в котором поиск вывода идет от цели к аксиомам и теоремам, истинность которых априорно известна. Чтобы определить выводимость формулы из посылок, надо найти формулы-предшественники, из которых нужная формула может быть выведена одним применением правила вывода. Затем по каждой из получившихся формул-предшественников, не являющейся аксиомой исчисления, определяется множество непосредственных формул-предшественников и т. д.

Дерево вывода получается, если все листья становятся аксиомами исчисления.

- *Продукционный вывод* — это метод, основанный на продукционных моделях. Такие модели отличаются от формальных продукционных систем тем, что они используют более богатые правила и содержат эвристическую информацию о специфике проблемной области, выражаемую, например, в виде семантических структур. В продукционном правиле есть элемент процедурности, т. к. предполагается, что правило будет использовано для выполнения некоторого действия. Декларативные знания не несут никакой информации о том, как они будут использованы.

  В общем случае продукционное правило может быть представлено в следующей форме: `(i); S; C; A=>B; Q`, где `A=>B` называют ядром продукции. Здесь:

  - `A` — совокупность посылок;
  - `B` — заключение;
  - `C` — условие применимости продукции;
  - `S` — сфера применимости продукции;
  - `(i)` — собственное имя продукции (метка);
  - `Q` — постусловие (если ядро реализовано, то может появиться необходимость изменения информации в базе данных, базе знаний или в продукционной системе, причем информация об этом может находиться в `Q`).

Вот лишь некоторые наиболее известные системы, применяющие механизм выводов [Попов 1987], [Гаврилова, Хорошевский 2001]:

- MYCIN (разработка Стэнфордского университета (**http://www.stanford.edu**)) — экспертная система в области медицины;
- PROSPECTOR (компании SRI International) — экспертная система в области геологии;
- Cattell (разработка компьютерного центра РАН (**http://www.ccas.ru/**)) — экспертная система в области социальной психологии.

5.3.5.3. Неточный вывод на знаниях

На конкурсе женской логики победил генератор случайных чисел.

Мужской фольклор

Данные и знания в экспертных системах могут иметь неопределенность, причем для ее описания применяются разнообразные средства.

Рассмотрим один из типов неопределенности — неточность. *Неточное высказывание* — это высказывание, истинность которого не может быть установлена

с определенностью. Модель оперирования с неточными данными и знаниями включает в себя две составляющие.

- Язык представления неточности.
- Механизм вывода на неточных знаниях.

Для построения языка необходимо выбрать форму представления неточности. Это может быть скаляр, интервал, распределение, лингвистическое выражение или множество. Также надо предусмотреть возможность приписывания всем высказываниям меры неточности. Из подходов к описанию неопределенности можно выделить вероятностный и алгебраический. Механизмы оперирования с неточными высказываниями можно разделить на два типа.

- *Присоединенный механизм*, сопровождающий процесс вывода на точных высказываниях. При этом для вывода на продукционных правилах надо определить следующие функции пересчета:
  - меру неточности посылки правила по мерам неточностей составляющих его высказываний;
  - меру неточности заключения правила по мерам неточности правила и посылки правила;
  - объединенную меру неточности высказывания, полученного из нескольких правил.
- *Специально-ориентированный механизм*, для которого характерно наличие схем вывода, настроенных на используемый язык представления неточностей.

В качестве примеров систем, применяющих аппарат неточного вывода на знаниях, приведем:

- CubiCalc (компании HyperLogic (**http://www.hyperlogic.com**));
- FuziCalc (компании FuziWare, Inc.).

5.3.6. Электронные библиотеки и инструментарий Интернета

5.3.6.1. Парадигма усиления информации

Рассмотрим парадигму усиления информации, которая достаточно удачно описывает роль ресурсов Интернета (рис. 5.6).

Парадигма проводит параллели с принципом, положенным в основу работы транзистора: усиление сигнала при подпитке от базы.

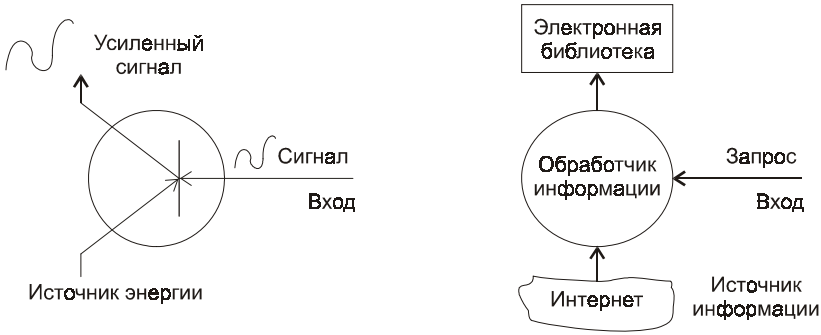


Рис. 5.6. Парадигма усиления информации

Интернет играет роль "информационного усилителя", питая пользовательский запрос информацией и переводя его в совокупность структурированной информации (электронную библиотеку).

5.3.6.2. Профессиональный поиск информации

Результаты профессионального поиска информации играют огромную роль в методах научного познания. Этапы, соответствующие началу научного исследования, могут быть алгоритмизированы.

- Формулировка проблемы, постановка цели (что хотим достичь) и конкретных задач (что необходимо сделать для достижения цели).
- Выбор методов исследования и построение стратегии информационно-аналитического поиска. Напомним, что общие методы научного познания обычно делят на следующие три большие группы:
  - методы эмпирического исследования (наблюдение, сравнение и др.);
  - методы, используемые как на эмпирическом, так и на теоретическом уровне исследования (абстрагирование, анализ и синтез и др.);
  - методы теоретического исследования (восхождение от абстрактного к конкретному и др.).

Как мы видим, практически все методы требуют поиска информации и использования результатов поиска в качестве входных данных.

В случае простейшей (и наиболее распространенной) стратегии поиска информации могут быть выделены два основных этапа.

- Запрос информации с целью исследования некоторой предметной области.
- Выборка действительно необходимой информации и генерация отчета по исследуемой проблеме.

Именно на эти этапы должны быть ориентированы системы, предназначенные для обработки информации, полученной из Интернета.

В настоящее время крупнейшим информационным ресурсом является Интернет, и в первую очередь среда World Wide Web (WWW) — глобальная интерактивная распределенная гипертекстовая информационная система. Предпринятые попытки оценить размер информации в Интернете согласуются в том, что в среде WWW содержится более миллиарда страниц.

Если предположить, что размер среднестатистической страницы составляет 4—10 Кбайт, то речь идет о терабайтах информации. Кроме поверхностной (не скрытой за поисковыми формами) части Интернета, существует скрытая (hidden, deep) часть, к которой относится множество крупных баз данных. Объем скрытой части оценивают в 500 раз больше, чем поверхностной.

Естественной моделью представления WWW является ориентированный граф, в котором вершины соответствуют страницам, а ребра — соединяющим страницы гиперссылкам. В рамках этой модели задача анализа структуры связей между страницами — это задача анализа структуры графа. WWW — это не просто месиво из случайных ссылок. Более 90 % страниц в WWW относятся к одной компоненте связности, в которой можно выделить четыре примерно одинаковые по размеру части [Broder 2000] (рис. 5.7). Ядро этой компоненты связности составляет компонента сильной связности, где для любой пары документов существует путь от первого документа ко второму. К ней примыкают части "исток" и "сток", в которых для любой страницы существует путь соответственно "в" или "из" компоненты сильной связности, но не наоборот. Отметим, что "исток" и "сток" могут быть связаны путями в обход компоненты. Остальные страницы относятся к так называемым "отросткам", для них не существует пути, соединяющего их с компонентой сильной связности. Прямым следствием этого наблюдения является, например, ограничение сверху на количество страниц WWW, которые можно посетить, начав с некоторой выбранной страницы и переходя по обнаруженным ссылкам.

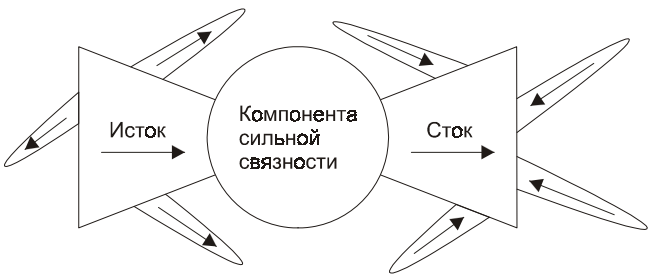


Рис. 5.7. Структура WWW

5.3.6.3. Проблемы работы с информационными ресурсами Интернета

Можно предположить, что в Интернете можно найти информацию практически по любой тематике. Однако сделать это достаточно сложно, т. к. наиболее распространенные и известные поисковые инструменты — справочники и поисковые серверы — не позволяют эффективно структурировать результаты поиска. Кроме того, возникает задача отсеивания данных, т. е. отсечения ненужной и несвязной информации от той ее части, которая будет полезной. Существует несколько проблем, возникающих при работе с информацией в Интернете и препятствующих быстрому и качественному поиску.

- *Недостаточная структурированность информации* (наличие так называемых слабоструктурированных данных). Для облегчения поиска в таких документах было предложено несколько моделей представления такой информации, в том числе Stanford's Object Exchange Model (**http://dbpubs.stanford.edu:8090/aux/index-en.html**), в рамках которой данные представляются в виде направленного графа с поименованными вершинами и дугами.
- *Избыточность информации*. Около трети информации в Интернете является точной или приблизительной копией других документов.
- *Наличие противоречивых и недостоверных сведений*. Типичным примером является включение некорректных ключевых слов в группу инструкций META, специально предназначенных для описания и индексирования документов поисковыми машинами. Некоторые поисковые машины сделали попытку решить эту проблему с помощью полнотекстовой индексации документов. Однако сетевые мастера Интернета предложили сразу несколько способов обмануть поисковый робот, например писать белым шрифтом по белому фону все то, что ранее включалось в инструкцию META.
- *Большое количество ошибок* (например, опечаток, грамматических ошибок, ошибок оцифровки). Эти ошибки являются следствием неконтролируемого качества, включая отсутствие редакторского контроля над публикуемой информацией.

> ***О качестве информационных ресурсов Интернета***
>
> Исходя из перечисленных фактов, ряд авторов называет Интернет "всемирной информационной помойкой", откуда становится сложно или даже невозможно извлечь нужную качественную информацию. Это достаточно жесткое утверждение, впрочем, становящееся все более и более справедливым.

Укажем на ряд особенностей поиска в Интернете, связанных с понятием человеческий фактор [Jansen, Sprink, Saracevic 2000].

- Поведение пользователя. Пользователь не готов долго ждать результата и не готов даже искать его в предоставленной поисковой системой выборке.

Практически половина пользователей не идет далее первого экрана, предложенного информационно-поисковой системой.

- Неумение делать запросы. Большинство пользователей не использует расширенные возможности поиска, такие как логические выражения. Кроме того, типичные поисковые запросы очень коротки — более 60 % запросов состоят всего из 1—2 слов.

Существует необходимость в совершенствовании программного обеспечения, которое поможет пользователю в интеллектуальном поиске и отборе нужной информации.

5.3.6.4. Коллекции информационных ресурсов в Интернете

Электронная библиотека — это система, обеспечивающая сообществу пользователей доступ (понятным для них образом) к большим репозиториям мультимедийной информации и знаний. Причем эти репозитории организованы при отсутствии каких-либо сведений о способах их применения [Когаловский 1999].

Коллекция информационных ресурсов — это систематизированная совокупность информационных ресурсов, объединенных по какому-либо критерию принадлежности, например по общности содержания, источников, назначения, по кругу пользователей, способу доступа и т. д. Коллекции являются наиболее распространенной формой организации информационных ресурсов в электронных библиотеках.

С функциональной точки зрения информационные ресурсы коллекции подразделяются на данные (информацию) и метаданные (метаинформацию). Ресурсы первого вида представляют собой интересующие пользователей сведения о предметной области этой коллекции. В свою очередь, метаданные коллекции характеризуют свойства самой коллекции и принадлежащих ей ресурсов в целом как сущностей реального мира.

Систематизация коллекций — задание структуры коллекции, осуществляемое на основе свойств ее предметной области и (или) свойств составляющих ее информационных ресурсов.

Систематизированный характер информационных ресурсов коллекции является принципиально важным ее свойством, отличающим коллекцию от других наборов таких ресурсов. Осмысленная систематизация информационных ресурсов не только облегчает доступ пользователей к ним, но и дает возможность целенаправленно и рациональным образом исследовать с их помощью предметную область коллекции.

Наряду с систематизированностью, к числу важных свойств любой коллекции относятся ее назначение, характеристики происхождения, способ задания состава принадлежащих ей ресурсов и правила их именования, виды

используемых информационных технологий, характеристики представления информационных ресурсов и т. д.

Возможны различные подходы к заданию состава коллекций.

- *Явным образом* — непосредственно как совокупность принадлежащих ей информационных ресурсов или как список ссылок на них (например, URL в WWW).
- *Неявным образом* — путем спецификации в какой-либо форме критерия принадлежности информационного ресурса данной коллекции. Примером может служить задание коллекции на основе полнотекстовой документальной системы путем спецификации поискового запроса.

Систематизация коллекции с использованием свойств ее предметной области должна основываться на концептуализации предметной области. Концептуальная модель предметной области может иметь различные формы представления. Если коллекция реализуется на основе технологии баз данных, то модель представляется в виде концептуальной схемы, которая отображается в среду СУБД и описывается схемой базы данных.

Для многих коллекций концептуальная модель предметной области представляется в форме классификатора, определяющего одномерное или многомерное пространство классификационных признаков. В таком случае отдельные точки или гиперплоскости этого пространства представляют собой классы сущностей предметной области, соответствующие ресурсам коллекции.

В качестве классификационных признаков обычно используют наиболее существенные свойства этих сущностей. Для классификационных признаков может быть предопределено множество их возможных значений. При этом множество значений отдельного признака может иметь линейную или иерархическую структуру.

5.3.6.5. Базы данных в Интернете

Среди информационных ресурсов Интернета особый интерес вызывают базы данных, которые подразделяются на [Краснослободцев, Смирнов, Лиходедов 1998]:

- текстовые (полнотекстовые, реферативные, библиографические);
- содержащие изображения и использующие средства мультимедиа;
- числовые и табличные;
- содержащие программное обеспечение.

Базы данных в Интернете рассчитаны как на массового, так и на профессионального потребителя. Наиболее часто используются базы данных первого типа. Ко второй группе относятся профессиональные информационные системы, представляющие собой специализированные базы данных и поисковые программы. Например, крупнейший мировой продавец информации,

компания Questel-Orbit, разместила в Интернете базу данных патентов (**http://plpatprd.questel.fr/plpat/jsp/en/login.jsp**). Специализированные базы данных, как правило, имеют свой узкоспециальный интерфейс и собственную уникальную структуру. В последние годы появилась тенденция к интеграции и стандартизации профессиональных баз данных.

5.3.6.6. Краткая история поисковых средств Интернета

Скажи мне – что ты ищешь в Интернете, и я скажу – кто ты.

Интернетовский фольклор

Активное внедрение средств компьютерных телекоммуникаций, глобальных компьютерных сетей и сетевых баз данных сделало сеть Интернет доступной и необходимой для многих. Интернет предоставляет своим пользователям информационные ресурсы и средства работы с ними.

Необходимость создания поисковых средств Интернета была осознана практически с момента создания сети. Исторически известны такие приложения, как Archer (для ftp), Veronica (для Gopher) и WAIS (поиск в индексированных базах данных). На сегодняшний день наиболее развитый и удобный в использовании инструмент — поисковые системы [Некрестьянов, Пантелеева 2002]. *Поисковая система* состоит из компонентов (рис. 5.8), таких как:

- *сетевой робот*, занимающийся сбором информации о доступных в Интернете ресурсах. Собранная информация складывается в хранилище. Его содержимое определяет набор документов, по которым идет поиск;
- *модуль индексирования*, создающий индексные структуры, по которым для достижения приемлемой эффективности и производится поиск;
- *поисковая машина*, получающая и выполняющая запросы пользователей.

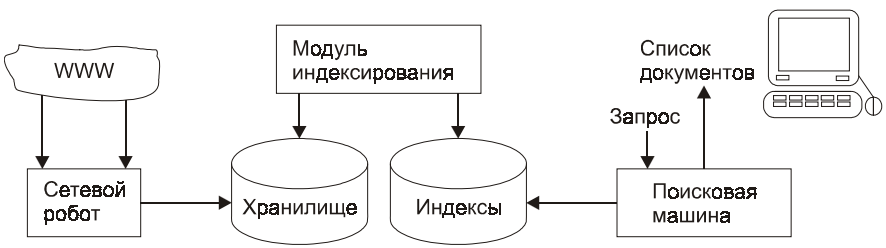


Рис. 5.8. Компоненты поисковой системы

Серверы, на которых расположены поисковые системы, называют *поисковыми серверами*. В число наиболее посещаемых англоязычных поисковых систем (и, соответственно, серверов) входят:

- Google (**http://www.google.com**);

- Yahoo (**http://www.yahoo.com**);
- AltaVista (**http://www.altavista.com**).

В русскоязычной части Интернета наиболее популярны:

- Яndex (**http://www.yandex.ru**);
- Rambler (**http://www.rambler.ru**).

Однако эти поисковые инструменты не позволяют эффективно структурировать результаты поиска. Кроме того, возникает задача отсеивания полученных данных, т. е. отсечения ненужной и несвязной информации от той ее части, которая будет полезной *(см. разд. 5.3.6.3)*.

Одним из наиболее перспективных методов работы с информацией в Интернете является ее структурирование в тематические электронные библиотеки. Активное использование персональных электронных библиотек должно служить созданию эффективной инфраструктуры для поддержки научных исследований и других сфер деятельности.

5.3.6.7. Искусственный интеллект и задача поиска в Интернете

Тематики исследований в двух крупных направлениях информатики — области искусственного интеллекта и Интернете — сближаются. Методы искусственного интеллекта все в большей степени ориентируются на задачи практического применения, а Интернет стремится к более сложным приложениям, требующим интеллектуального поведения.

Многие методы и приемы искусственного интеллекта нашли применение в конкретных проблемных областях. Перечислим лишь некоторые направления, которые могут с успехом применяться для разработки интернет-приложений.

- Обработка естественного языка.
- Извлечение информации из баз данных.
- Экспертные консультирующие системы.

Понятие *онтология* было введено в искусственный интеллект Томом Грубером [Gruber 1993]. Подобно формальным спецификациям программных продуктов, онтология — это описание концептов и связей между ними. Предполагалось, что онтологии будут использованы исключительно для облегчения взаимодействия интеллектуальных агентов. Затем понятие онтологии было расширено. В частности, иерархическая структура классов в объектно-ориентированном программировании представляет собой онтологию.

Большую роль в сборе информации о существующих ресурсах Интернета играют сетевые роботы, являющиеся интеллектуальными агентами. Они, начиная с некоторого множества ссылок на страницы в сети Интернет,

рекурсивно обходят ресурсы, извлекая ссылки на новые ресурсы из полученных документов, до тех пор, пока не будет выполнено некоторое условие остановки. Уточним понятия агента и интеллектуального агента.

- *Агент* — это сущность, находящаяся в некоторой среде, от которой она получает данные, отражающие происходящие в этой среде события. Агент интерпретирует данные и исполняет команды, которые воздействуют на среду.
- *Интеллектуальный агент* — программно или аппаратно реализованная система, обладающая следующими свойствами:
  - *автономность* — способность функционировать без вмешательства человека;
  - *общественное поведение* — способность функционировать в сообществе с другими агентами;
  - *реактивность* — способность своевременно отвечать на изменения окружающей среды;
  - *проактивность* — способность агента брать на себя инициативу.

В настоящее время исследование применимости идей искусственного интеллекта к среде Интернет ведется в различных направлениях [Mahalingam, Huhns 1997].

- Обработка поисковыми машинами запросов на естественном языке. Система получает запрос на естественном языке, с помощью грамматических и лингвистических правил сопоставляет запрос с информацией в Интернете. Удачным примером такой системы является Яndex (**http://www.yandex.ru**). В ней независимо от того, в какой форме употреблено в запросе слово, поиск учитывает все его формы по правилам русского языка.
- Продукционно-эвристическое распознавание естественного языка. Системы, развивающиеся в этом направлении, как правило, являются узкопрофильными и предназначены для выполнения очень узко сформированных целей или обработки ограниченных областей данных. Пример системы — FAQFinder (**http://www.mines.edu/academic/computer/faq/**).
- Автоматическое наполнение базы знаний. Это и следующее направления используют агентную технологию. Данное направление реализовано в интеллектуальном браузере WebWatcher (**http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/theo-6/web-gent/www/project-home.html**).
- Использование эвристических правил для установки приоритета. Это направление реализовано в другом интеллектуальном браузере — Letizia (**http://lcs.www.media.mit.edu/people/lieber/Lieberary/Letizia/Letizia-Intro.html**).

- Направление нейронных сетей. Основой здесь являются сети искусственных нейронов и других аналогичных конструкций, присущих нервной системе человека, в которых протекают психические процессы, опирающиеся на физиологический и биохимический уровни. Пример реализации этого направления — система работы со знаниями Autonomy (**http://www.autonomy.com/**).
- Принцип разделяемых знаний. Реализуется в виртуальных группах, т. е. организациях и рабочих коллективах людей, которые работают и общаются между собой в интерактивном режиме. Знакомство с такого рода системами можно начать с AIMS (**http://www.eimc.brad.ac.uk/~dhobbs/vle/aims.htm**).

Экспериментальные системы интеллектуального поиска используют одновременно мощь существующих поисковых систем и элементы искусственного интеллекта для отбора и анализа извлеченной из Интернета информации [Холчева 1999]. Поисковая система по запросу пользователя обращается на крупнейшие поисковые серверы. Полученные ссылки сортируются. Анализируются HTML-файлы, расположенные по этим адресам. Пользователю предлагаются выдержки из текста документов, отражающие результаты поиска.

Каждый пользовательский запрос, прежде всего, преобразуется в дизъюнктивную форму со строго определенным порядком следования логических операций. Приоритеты операций расставлены следующим образом (табл. 5.1).

Таблица 5.1. *Приоритеты логических операций*

Приоритет	Обозначение	Назначение
0	" "	Поиск по точному соответствию
1	!	Исключение лексемы
2	&	Логическое И
	|	Логическое ИЛИ

Текстовый редактор предполагает возможность фильтрации и сортировки логических абзацев. Сортировка может производиться по следующим принципам:

- по релевантности логических абзацев;
- по приоритету документов;
- по рассматриваемым поисковым серверам;
- по частоте появления документа на серверах;
- по заголовкам документов.

Таким образом, на вход генератора итогового документа подается набор логических абзацев со списками атрибутов. Синтезировать отчет можно в нескольких формах, различающихся порядком следования логических абзацев, с предоставлением пользователю возможности переключаться из одного режима в другой.

5.4. Инструментарий поддержки процессов некоторых технологических подходов

5.4.1. Системы верификации программ

Поддержка технологий *верификации программ* осуществляется специальными системами. Большинство из них являются научно-исследовательскими и не имеют коммерческого применения. Одной из наиболее интересных современных работ в области формальных подходов верификации программ является B-технология. На ее основе была осуществлена разработка системы управления парижским метрополитеном "METEOR".

5.4.2. Системы формального преобразования программ

Цель создания *средств формального преобразования программ* состояла в том, чтобы автоматизировать задачи разработки программного обеспечения, модификацию программ и исправление ошибок. Преобразования специфицировались, используя правила, в которых левая часть — это образец исходного кода, а правая — действия, которые должны быть выполнены, если левая часть встретилась в тексте.

В качестве примера рассмотрим систему TXL [Kozaczynsky, Hing, Engberts 1992], которая конвертирует код, написанный на диалекте, в код на базовом языке. Это выполняется через преобразование синтаксического дерева грамматики диалекта в синтаксическое дерево базового языка, а затем получение новой программы из последнего. Для преобразования деревьев использовался рекурсивный алгоритм обхода дерева.

Системы формального преобразования программ по многим идеям близки к системам, используемым при сопровождении программ, и системам работы с текстами.

5.4.3. Средства сборочного программирования

Несмотря на естественность и популярность сборочного программирования, средства его поддержки практически отсутствуют или не выходят за пределы компаний по разработке программного обеспечения. Это объясняется тем, что с одной стороны существует огромное количество правовых проблем,

препятствующих свободному копированию программных модулей, а с другой — низким качеством и плохой документированностью таких программ. В настоящее время наиболее популярным средством сборочного программирования являются репозитории, которые мы рассмотрим *в разд. 5.5.4*.

5.5. Инструментальные системы

Целое — это больше, чем сумма частей.
Аристотель

Инструментальной системой будем называть некоторую логически связанную совокупность программных инструментов, поддерживающую разработку и сопровождение программных систем на данном языке программирования или ориентированных на какую-либо конкретную предметную область.

Эту группу характеризует в первую очередь интеграция нескольких инструментов под единой оболочкой и использование общего репозитория.

Выделяют три группы инструментальных систем.

- Инструментальные среды программирования.
- Средства автоматизации разработки программ.
- Интегрированные среды.

В идеальном варианте инструментальные системы должны распространяться на максимально возможное количество процессов и покрывать максимум стадий жизненного цикла. Однако исторически сложилось так, что инструментальные среды в большей степени связаны с процессами программирования, тестирования и отладки, а средства автоматизации разработки программ — с анализом и проектированием, т. е. с более творческими процессами (рис. 5.9).

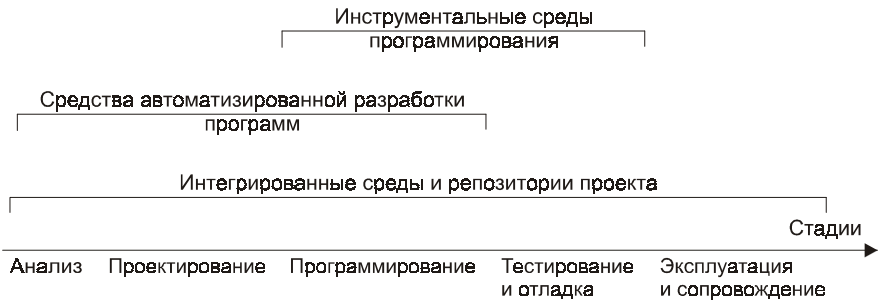


Рис. 5.9. Инструментальные системы

5.5.1. Инструментальные среды программирования

Инструментальные среды программирования обычно содержат текстовый редактор, компилятор, отладчик и средства подсказки. Кроме того, в них могут быть включены и другие инструменты, позволяющие выполнять статический

и динамический анализ программ. Эти инструменты взаимодействуют между собой через обычные файлы с помощью стандартных возможностей файловой системы.

Различают среды общего назначения и языково-ориентированные среды.

- *Среды общего назначения* содержат набор программных инструментов (например, текстовый редактор, редактор связей и т. п.), позволяющих выполнять разработку программ на разных языках программирования. Для программирования на конкретном языке программирования требуются дополнительные инструменты, ориентированные на этот язык.
- *Языково-ориентированные среды* предназначены для поддержки разработки программ на каком-либо одном языке программирования, причем построение такой среды базируется на знаниях об этом языке.

Инструментальная среда не обязательно должна функционировать на том компьютере, на котором должен будет применяться разрабатываемый с ее помощью программный продукт.

Инструментальные среды программирования характерны тем, что:

- поддерживают различные методологии;
- применяются в различных технологиях;
- применяются командами, работающими над различными проектами;
- используются для разработки разнообразных приложений;
- разрабатываются одной компанией. При этом возможность интеграции инструментов других компаний отсутствует.

Примеры инструментальных сред следующие:

- Visual Studio (компании Microsoft (**http://www.microsoft.com**));
- Forte for Solaris Developer Tools (компании Sun Microsystems, Inc. (**http://www.sun.com**));
- Delphi Suite (компании Borland Software Corporation (**http://www.borland.com**)).

5.5.2. Средства автоматизации разработки программ (CASE-средства)

Средства автоматизации разработки программ (Computer-Aided Software Engineering — CASE) — это инструментарий для системных аналитиков, разработчиков и программистов, позволяющий автоматизировать процесс проектирования и разработки программного обеспечения. Первоначально под CASE-средствами понимались средства, применяемые на ранних процессах жизненного цикла, и в первую очередь — на наиболее трудоемких процессах анализа и проектирования. Международный стандарт [ISO/IEC 14102:1995]

определяет CASE-средства более широко — как программное средство, поддерживающее процессы жизненного цикла программного обеспечения. CASE-средства характеризуются наличием мощных средств визуального моделирования.

Средства автоматизации разработки программ характерны тем, что:

- поддерживают единственную методологию;
- ориентируются на определенную технологию;
- предназначаются для команд, работающих над единственным проектом (так сложилось исторически);
- используются для разработки информационных систем;
- разрабатываются одной компанией. Возможность интеграции инструментов других компаний отсутствует.

Примеры CASE-средств следующие:

- Oracle Designer (компании Oracle (**http://www.oracle.com**));
- ERwin (компании Computer Associates International, Inc. (**http://ca.com/**));
- Rational Rose (компании Rational Software Corporation (**http://www-306.ibm.com/software/rational/**)).

5.5.3. Интегрированные среды

Интегрированная среда — это совокупность программных инструментов, поддерживающая все процессы жизненного цикла программного обеспечения в рамках определенной технологии. Компонентами интегрированных сред являются:

- инструменты управления процессами;
- инструменты управления проектом;
- инструменты конфигурационного управления;
- инструменты верификации;
- инструменты поддержки разработки документации.

Выделяют три уровня интеграции инструментов в интегрированных средах.

- Уровень 1 — Интеграция инструментов очень слабая. Как правило, обмен информацией между ними происходит через интерфейсы экспорта и импорта.
- Уровень 2 — Интеграция инструментов одной компании осуществляется на основе единого репозитория. Интеграция инструментов других компаний с этими инструментами происходит по образцу уровня 1.
- Уровень 3 — Интеграция всех инструментов осуществляется с помощью общего репозитория. При этом любой инструмент любой компании

может осуществлять взаимодействие через службы взаимодействия с репозиторием.

Интегрированные среды характерны тем, что:

- поддерживают различные методологии;
- определяют технологию разработки;
- применяются командами, работающими вместе над несколькими проектами;
- как правило, используются для разработки научных и инженерных приложений;
- разрабатываются одной компанией, но имеется возможность интеграции инструментов других компаний.

Примеры интегрированных сред следующие:

- WebSphere Studio WorkBench (компании IBM (**http://www-306.ibm.com/software/awdtools/workbench/**));
- CohesionWorX (компании Digital Equipment Corp.);
- SorfBench (компании Hewlett-Packard (**http://www.hp.com**)).

> ***Об аппаратной платформе интегрированных сред***
>
> Обратим внимание, что интегрированные среды практически отсутствуют на персональных компьютерах. Это можно объяснить сильным монополизмом компаний разработчиков инструментария на рынке персональных компьютеров и их нежеланием предоставлять возможность интеграции инструментов других компаний.

5.5.4. Репозитории проекта

Репозиторий — это приложение, обеспечивающее хранение и обработку данных и метаданных, а также их предоставление по запросам. Филипп Бернштейн (Philipp Bernstein) определяет репозиторий как разделяемую базу данных с информацией об артефактах проектирования, требующую некоторых дополнительных функций управления помимо предоставляемых обычными системами баз данных. Репозитории приобретают роль фундамента всей информационной среды и связанных с ней систем программирования (компиляторов, отладчиков, анализаторов и т. п.).

На одной из конференций по программной инженерии отмечалось, что около 80 % программного обеспечения создается на базе уже имеющегося. Следовательно, следует иметь инфраструктуру, которая будет поддерживать архивы и достаточно интеллектуальный поиск нужных прототипов и фрагментов. *Повторное использование* (reuse) программных компонентов позволяет существенно сократить расходы на разработку программного обеспечения.

О необходимости быть внимательными, используя готовые объекты в своих программах

Одной из популярных фольклорных историй стала история с австралийскими программистами. Начальник отдела симуляций наземных операций приказал разработчикам смоделировать перемещения кенгуру и их реакцию на вертолеты. Будучи грамотными программистами, они использовали готовые программные объекты, описывающие поведение пехоты в аналогичной ситуации, заменив изображения солдат на изображения животных и увеличив их скорость. Во время демонстрационной симуляции летчики "разбудили" кенгуру, пройдя над ними на малой высоте. Кенгуру разбежались, как и предполагалось, и наблюдатели понимающе кивнули... А затем сильно удивились, т. к. кенгуру, перегруппировавшись, появились из-за холма и выпустили тучу стингеров по злополучным вертолетам.

Репозиторий дает еще один повод поговорить о классификациях коллекций информационных ресурсов. Любой достаточно крупный проект в области разработки программного обеспечения накапливает вокруг себя огромное количество текстов спецификаций, программных кодов, документации и т. п. В идеальном случае для хранения такой информации и работы с ней удобно применять системы работы с коллекциями электронных информационных ресурсов программных компонентов. Такие коллекции важны для поддержки хорошо известных в программировании парадигм повторного использования и сборочного программирования, используемых в процессе создания и сопровождения программного обеспечения. Данная область исследований находится в тесном соприкосновении с такими направлениями как компонентное программирование, электронные библиотеки, интегрированные системы, искусственный интеллект.

Общая задача формирования информации для разработки программного обеспечения, извлечения ее из всевозможных хранилищ и формирования на ее основе программных продуктов может быть рассмотрена на двух уровнях.

- Глобальном — как структуризация информации и систематизация моделей (в том числе, алгоритмов) и знаний в области профессионального программирования. Это уровень систематизации, полезный для понимания широты и глубины всей предметной области.
- Локальном — как систематизация информации и фрагментов программ для быстрой сборки результирующего программного продукта. Данный уровень направлен на программистов-профессионалов, непосредственно занятых разработкой.

В апреле 1994 года компании Instruments и Microsoft решили объединить свои усилия в создании репозитория метаданных. Целью этого сотрудничества была разработка для обеих компаний репозитория следующего поколения, который они могли бы интегрировать в инструментарий своих программных продуктов. Результатом работы стал Microsoft Repository (компании Microsoft (**http://www.microsoft.com**)). Однако в 2001 году по коммерческим соображениям проект был прекращен. В некоторой степени его заменили такие продукты и технологии, как ADO.NET и утилита Object Browser.

Работа с информацией в области профессионального программирования имеет свои существенные особенности, которые отражены на рис. 5.10. Каждый компонент занимает свое место в трехмерном пространстве, причем каждое из измерений играет свою существенную роль.

- С одной стороны, информация имеет различную степень формализации — от первичной текстовой спецификации к модельному представлению и алгоритму на языке программирования.
- С другой стороны, структурированность информации может быть как сильной (в существующих репозиториях проектов), так и слабой (на сайтах в Интернете).
- И, наконец, способы представления информации могут включать информационные объекты, тексты и бинарный код.

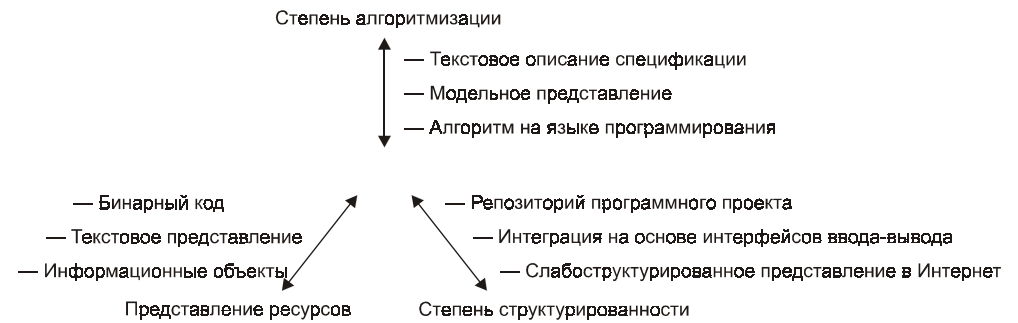


Рис. 5.10. Особенности коллекций ресурсов в области программирования

Существуют многочисленные проблемы, связанные со спецификой разработки программного обеспечения и психологией работы программистов. Вот некоторые из них, серьезно препятствующие созданию компонент, пригодных для использования в коллекциях:

- недостаточная продуманность методов классификации программных компонентов;
- немногочисленность технологических подходов, использующих электронные коллекции программных компонентов;
- стремление многих программистов начать разработку "с нуля", без использования чужих фрагментов и наработок.

Возможная структура систем работы с коллекциями электронных информационных ресурсов программных компонентов представлена на рис. 5.11.

Пользователь взаимодействует с системой на трех уровнях, причем каждый поддерживается редактором (или семейством редакторов):

- онтологическим, позволяющим вводить и изменять онтологическое представление избранной предметной области;

- компоненто-ориентированным, позволяющим структурировать компоненты и определять интерфейсы компонент с последующим помещением их в собственный репозиторий системы;
- синтаксически-ориентированным, позволяющим работать с фрагментами кода на некотором языке программирования.

Базовая часть системы включает:

- ядро, реализующее основные алгоритмы работы с информацией;
- подсистему настраиваемого интерфейса, взаимодействующую с многочисленными источниками информации;
- заменяемые редактируемые компоненты — онтологическое представление и репозиторий.

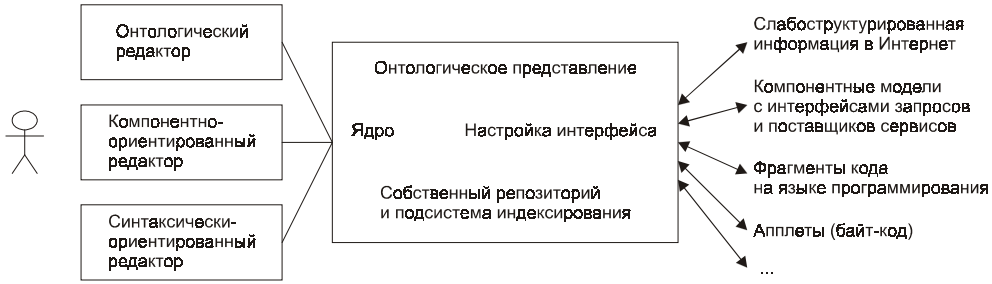


Рис. 5.11. Компоненты систем работы с коллекциями

Обратим внимание на то, что практически все технологические подходы разработки программного обеспечения предполагают работу с тремя основными типами информации: модельными спецификациями, интерфейсом прикладного программиста и окружением проекта. Выделим три уровня репозиториев, соответствующих этим типам.

- Уровень моделирования достаточно хорошо может быть описан универсальным языком UML. Данный язык является абстрактным, не привязанным к конкретной модели. Язык дает возможность описать зависимости элементов, иерархию, взаимосвязи, свойства и т. п.
- Уровень программного интерфейса разумно описывать с помощью языка определения интерфейсов IDL, обеспечивающего независимость спецификации интерфейсов от их реализации. Уровень играет не только роль промежуточного слоя, но его средства поддерживают распределенное программирование (в том числе в Интернете).

- Репозиторий окружения предназначен для хранения информации, разделяемой компонентами и подкомпонентами систем программирования в процессе их работы. Приведем основные группы и подгруппы полезной информации:
  - языково-независимая группа (информация для отладчика, информация для просмотрщика исходных текстов);
  - языково-зависимая группа (например, для языка C++ информация для шаблонов, коды инлайн-функций, виртуальные функции);
  - группа контроля репозитория (контроль версионности, контроль отношения к проекту, тип параллельной обработки, тип управления репозиторием, проверка цифровой подписи).

5.6. Средства поддержки коллективной разработки

Коллективные разработки часто сталкиваются с проблемами разделения ресурсов (файлов и каталогов) и необходимости синхронизации времени и места.

5.6.1. Системы разделения файлов

Для поддержки коллективной работы с файлами применяются три основных класса систем.

- Системы управления версиями файлов.
- Системы управления пространствами пользователей.
- Системы синхронизации удаленных пространств.

Если три класса систем выпускаются одним производителем, то часто каждая последующая система в этом списке использует предыдущую систему, выступая в качестве надстройки.

> ***О системах управления версиями***
>
> Обратим внимание, что системы управления версиями файлов следует использовать не только при коллективной, но и при индивидуальной (авторской) разработке для решения задачи отслеживания изменений между различными версиями. Мы помещаем информацию об этих системах в данном разделе лишь с целью подчеркнуть их полную незаменимость при коллективных разработках.

Поскольку данные системы действительно очень важны, их реализации существуют в большом количестве практически для всех операционных систем. Приведем два стека таких систем в табл. 5.2.

Таблица 5.2. *Примеры средств поддержки коллективной разработки*

Классы систем	Среда операционной системы Windows	Среда операционной системы Unix
Управления версиями файлов	RCS (Revision Control System) (**http://www.gnu.org/software/rcs/rcs.html**)	SCCS (Source Code Control System) (**http://www.sun.com**)
Управления пространствами пользователей	CVS (Concurrent Versions System) (**http://www.cvshome.org**)	bringover/putback (**http://www.sun.com**)
Синхронизации удаленных пространств	Rsync (**http://samba.anu.edu.au/rsync/**)	synctool (**http://www.sun.com**)

5.6.1.1. Система управления версиями файлов

Множество грехов отпадает, если рядом с колеблющимся
стоит свидетель его поступков.
Эпикур

Система управления версиями файлов предназначена для отслеживания изменений между различными версиями файлов и разделения доступа к ним. Файлы обычно содержат программы на языках программирования и тексты.

Проиллюстрируем работу с версиями на примере системы SCCS. При передаче файла под ее управление, система создает новый файл (так называемый `s.`-файл) в подкаталоге с именем SCCS. Именно в этом дополнительном файле будут отслеживаться и храниться все изменения:

```
/имя_файла/SCCS/s.имя_файла
```

Укажем операции, наиболее часто выполняемые над файлом, помещенным под систему управления версиями, и проиллюстрируем их работу на рис. 5.12.

- `sccs create <имя_файла>` — передать файл под управление SCCS.
- `sccs get <имя_файла>` — извлечь последнюю версию файла, открытую только для чтения.
- `sccs edit <имя_файла>` — извлечь последнюю версию для редактирования, открытую для записи, и заблокировать возможность открыть файл для редактирования остальным пользователям.
- `sccs unedit <имя_файла>` — отменить редактирование и исключить информацию о попытке редактирования, дать возможность открыть файл для редактирования остальным пользователям.
- `sccs delta <имя_файла>` — вернуть измененную версию и добавить комментарий.

- `sccs delget <имя_файла>` — комбинация команд `delta` и `get`.

Другие, наиболее часто применяемые операции выглядят так:

- `sccs prt <имя_файла>` — вывести протокол изменений файла, включая комментарии.
- `sccs diffs <имя_файла>` — выполнить сравнение двух версий файла.
- `sccs info <имя_файла>` — перечислить все файлы текущего каталога, открытые на редактирование командой `edit`.

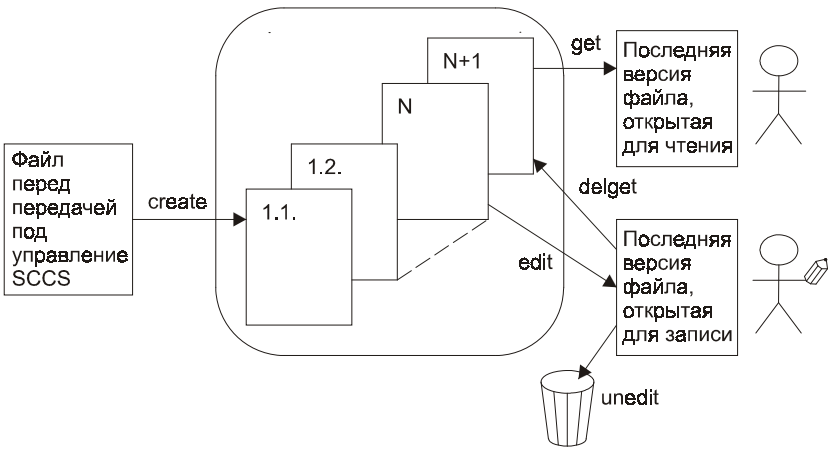


Рис. 5.12. Работа с системами управления версиями текстов

Обратим внимание на то, что системой отслеживаются именно изменения (дельты) между версиями, а не сами полные версии, что позволяет занимать небольшое количество дискового пространства. Каталог SCCS часто называют репозиторием проекта.

Как и все широко распространенные программы утилита, SCCS имеет надстройки в виде графического интерфейса, например утилиту vertool.

5.6.1.2. Система управления пространствами

Система управления рабочими пространствами пользователей предназначена для обмена результатами работы между отдельными разработчиками через объединение результатов в выделенном рабочем пространстве. Система работает на основе модели "копирование-изменение-слияние" и понятия "рабочее пространство".

Рабочее пространство — это специальным образом выделенная совокупность каталогов с файлами, переданными системе управления версиями файлов. Кстати, на профессиональном жаргоне разработчики называют свои рабочие пространства песочницами.

Две основные команды работы с этими пространствами (рис. 5.13) представляют собой копирование файлов: из пространства-предка в пространство-потомка (например, команда `bringover`) и обратно (например, команда `putback`). Обе эти команды имеют одинаковый набор параметров, позволяющих указать пространство-потомка (опция `-w`), пространство-предка (опция `-p`), а также список файлов и/или каталогов. Опция `-n` позволяет не выполнять реальное копирование, а лишь выдать информацию о возможных действиях.

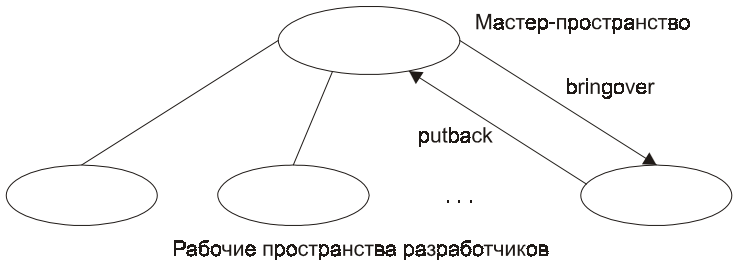


Рис. 5.13. Работа с пространствами

Модель "копирование-изменение-слияние" отражает схему организации работы с пространствами. Основная последовательность действий такова:

1. Создается основное мастер-пространство, от которого разработчики порождают свои рабочие пространства.
2. После внесения изменений в своем рабочем пространстве разработчик передает их в мастер-пространство.
3. Все разработчики периодически обновляют свои рабочие пространства из мастер-пространства.

Основная проблема, которая при этом может возникнуть, — это конфликт изменений в том случае, когда разработчик передает измененные файлы в мастер-пространство, но они уже оказываются измененными другим разработчиком. Можно попытаться избежать такой ситуации, выполняя обновления как можно чаще или установив регламент для внесений изменений.

Тем не менее, если такая ситуация возникла, то требуется выполнить под управлением разработчика слияние файлов, в ходе которого конфликт будет разрешен. Процесс слияния в большинстве случаев требует интеллектуального подхода при принятии решений в формировании итогового файла. Автоматизировать можно лишь сам процесс формирования файла, например, используя утилиту слияния файлов filemerge.

5.6.1.3. Система синхронизации удаленных пространств

В ряде случаев может оказаться, что пространства некоторых разработчиков находятся на значительном удалении друг от друга. Очень часто пространства находятся в различных локальных сетях, каждая из которых имеет выход в Интернет. При этом возникают две проблемы: *организация пересылки изменений* и их *синхронизация*.

Для осуществления пересылки можно воспользоваться стандартными средствами Интернета, например электронной почтой или утилитой ftp. Для облегчения синхронизации следует завести два дополнительных пространства-порта и выполнять их синхронизацию (например, при помощи семафоров). Утилиты, выполняющие пересылку, должны тщательно сверять контрольные суммы, чтобы убедиться в корректности полученной информации. Естественно, в пересылаемый пакет информации входят только инкрементальные изменения (дельты), а не измененные файлы целиком.

5.6.2. Системы поддержки работы виртуальных групп

Принцип разделяемых знаний реализуется в виртуальных группах, т. е. в организациях и группах людей, которые работают и общаются между собой в интерактивном режиме. Общение в виртуальных группах очень часто осуществляется через Интернет, а в качестве средств общения выступают традиционные и хорошо известные средства, такие как:

- видеоконференции;
- аудиоконференции;
- средства группового общения в реальном времени (чаты);
- средства группового общения с разным временем подключения (доски объявлений, форумы);
- электронная почта (включая списки рассылки, архивы и поиск по ним).

Однако все перечисленные средства имеют определенные недостатки. Например, видео- и аудиоконференции не позволяют передавать файлы, а средства группового общения в Интернете не обеспечивают секретности от доступа любых других пользователей, с одной стороны, и позволяют разработчикам выступать анонимно, с другой стороны.

Системы поддержки работы виртуальных групп должны хранить и обрабатывать информацию, а также помогать разработчикам осуществлять общение друг с другом. Сформулируем некоторые общие требования к таким системам.

- Необходимость регистрации пользователя при его входе в систему.
- Строго авторизованный доступ ко всем данным в системе.

- Сохранение всей переписки и прочих данных в архиве.
- Использование системы управления версиями текстов для программ и документации.

Классификация систем поддержки работы групп по предоставляемым ими возможностям общения была предложена Бобом Йохансеном (Bob Johansen).

- В одно и то же время, в одном и том же месте (same time, same place). Это классический случай, когда разработчики имеют возможность встречаться в одном помещении в определенное время. Здесь оказываются полезными следующие системы поддержки:
  - система упорядочивания доступа к данным для показов видеозаписей, слайдов и других документов;
  - система проведения мозговых штурмов и процедур принятия решений. Система дает возможность вносить идеи в список сразу после их возникновения. Как правило, такие системы обладают возможностями для быстрого голосования.
- В одно и то же время, в разных местах (same time, different place). Системы должны обеспечивать общение в реальном времени. Для этого обычно применяются средства аудио- и видеоконференций.
- В разное время, в одном и том же месте (different time, same place). Это довольно редкая ситуация, когда существует комната группы, но разработчики не имеют возможности собраться в ней в одно и то же время. Разработчику должна быть предоставлена возможность доступа к репозиторию проекта и всей информации, доступной на данном этапе проекта.
- В разное время, в разных местах (different time, different place). Системы реализуют возможности ведения конференций, форумов и чатов с асинхронным подключением, возможность доступа к репозиторию проекта.
- Неограниченные возможности доступа к данным (anytime, anyplace). Эта категория оставлена автором классификации на дальнейшее развитие коммуникационных технологий, когда возможности доступа к данным станут практически неограниченными.

Обратим внимание на то, что часто тип систем поддержки работы виртуальных групп зависит от рода их деятельности.

- *Группы управления*. Управляющие работники компаний, находящиеся в удаленных друг от друга подразделениях, обычно используют видеоконференции.
- *Группы проекта*. Разработчикам требуется максимально возможное количество средств общения и доступа к репозиториям.
- *Группы сопровождения*. Группам необходимы средства связи и быстрый доступ к данным из любого места.

Приведем несколько наиболее известных систем поддержки виртуальных групп:

- eRoom (компании Instinctive Technology, Inc.);
- Facilitate.com (компании Facilitate.com (**http://www.facilitate.com**));
- Consensus@nyWARE (компании Soft Bicycle (**http://c2.com/cgi/wiki?TcbWorks**));
- 3-2-1 IntraNet (компании Internet Media, Inc.).

5.7. Естественно-языковый интерфейс

Не надо очеловечивать компьютеры, они этого страсть как не любят!
Программистский фольклор

Системы, обрабатывающие естественно-языковые высказывания, можно разделить на следующие классы:

- диалоговые системы;
- интеллектуальные вопросно-ответные системы;
- автоматизированные обучающие системы и системы контроля знаний;
- системы обработки связных текстов.

5.7.1. Диалоговые системы

Когда много спрашивают — мало думают и плохо помнят.
Максим Горький

Программа "Элиза" Джозефа Вейценбаума (Jozeph Weizenbaum) — одна из первых программ, позволяющая вести диалог с пользователем в свободно конструируемой форме на различные темы. Программа ведет диалог, задавая вопросы. *Диалог* строится с учетом тематически объединенных рубрик. *Рубрика* — это некоторая смысловая единица, характеризующая определенный набор слов или словосочетаний по тематике задаваемого пользователю вопроса и составленная из слов и словосочетаний, встречающихся в ответах пользователя.

Рассмотрим пример ведения диалога. Программа предлагает поговорить по некоторым темам, которые выбираются пользователем из меню. Например, выбрана тема "Языки программирования". Программа начинает диалог:

```
"Давайте поговорим о языках программирования. Какому из языков программирования вы отдаете предпочтение?"
```

Первый вопрос выбирается из первой, начальной, рубрики или всех рубрик темы случайным образом. Текущая рубрика запоминается. Пользователь вводит ответ. Например:

```
"На работе я выполняю проект на языке C++, а дома люблю писать программы на языке РЕФАЛ".
```

Программа ищет в рубрикации ключевые слова из ответа пользователя, определяет на их основе, что сказал пользователь, и, основываясь просто на этих ключевых словах пользовательского ответа, случайно выбирает вопрос из текущей рубрики. Если в текущей рубрике не найдено ни одно ключевое слово, система ищет ключевые слова в других рубриках и при нахождении их меняет текущую рубрику. Например, если в какой-либо рубрике темы "Языки программирования" есть слово "C++", которому соответствует вопрос "Сколько лет вы уже программируете на C++?", то программа задает этот вопрос. Остальные ключевые слова, которым соответствуют какие-либо вопросы, запоминаются и ставятся в очередь. Основываясь на ключевых словах (в примере это были слова "C++" и "РЕФАЛ"), накапливается очередь вопросов.

Это пример контекстно-зависимого диалога. Программа, не основываясь на семантике, выбирает отдельные слова из пользовательского ответа. Этот анализ называют дескриптивно-рубрикационным. Недостатком такого подхода является то, что программа должна хранить огромное количество рубрик, поиск же по этим рубрикам занимает большое время. Такие системы не учитывают семантики и из-за этого могут неправильно отреагировать на ответ пользователя. К тому же, при таком подходе не может быть организован диалог, цель которого — принятие программой решений по его результатам, и, соответственно, значимость которого была бы достаточно высока. Такие системы применимы в качестве игрушки, демонстрирующей возможности компьютера. Программа "Элиза" имела именно такое назначение. Достоинства этого подхода в том, что система может вести разговор с пользователем практически на любую тему, на которую написаны соответствующие рубрики. Кстати, наиболее популярным было использование программы "Элиза" в качестве психоаналитика.

5.7.2. Вопросно-ответные системы

Суди о человеке больше по его вопросам, чем по его ответам.
Ф. Вольтер

Интеллектуальные вопросно-ответные системы предназначены для нахождения ответа на пользовательский вопрос. Системы такого класса, как правило, также основаны на рубриках. Отличие от диалоговых систем в том, что рубрикация представлена в виде дерева. Система ищет в пользовательском вопросе ключевые слова, которые соответствуют высшим рубрикам. Если в вопросе пользователя найдено ключевое слово, которое соответствует высшей рубрике, то эта рубрика становится текущей, и в вопросе ищется ключевое слово, которое соответствует рубрике следующего уровня. И так далее до тех пор, пока система не дойдет до последнего уровня. Если в вопросе пользователя не найдено ни одного ключевого слова высшего уровня, то система может задать уточняющий вопрос или же ответить, что ничего

подобного не найдено. Это пример более сложного, дескриптивно-рубрикационного подхода.

Достоинства дескриптивно-рубрикационного подхода в том, что рубрики более четко организованы иерархически, и анализ ответа получается более верным, чем в контекстно-зависимом диалоге, рассмотренном в предыдущем разделе. Однако такой подход имеет те же недостатки, что и в случае диалоговых систем.

5.7.3. Автоматизированные обучающие системы и системы контроля знаний

Плохому учатся и без учителя.
Античный афоризм

Автоматизированные обучающие системы предназначены для обучения пользователей какой-либо предметной области с последующей проверкой знаний. Наиболее эффективная форма проверки знаний — общение пользователя с компьютером на естественном языке, поэтому такие системы должны включать блоки анализа естественно-языковых высказываний. Рассмотрим наиболее интересные подходы к реализации семантических анализаторов высказываний в автоматизированных обучающих системах (АОС) [Искусственный 1990].

Семантический анализатор в АОС "Катехизис" был основан на разборе семантической маски ожидаемого высказывания. Здесь маска высказывания задавалась обратной польской записью. Причем в этой записи сначала были записаны негативные дескрипторы, т. е. те, которые не должны быть найдены в ответе пользователя, а потом позитивные. На введенной пользователем строке высказывания разбиралась одна семантическая маска. Если высказывание в целом не соответствовало маске, то оно считалось неверным в контексте заданного вопроса.

Разработанная в начале 80-х годов XX века "ОН Микро" была одной из первых автоматизированных обучающих систем, имеющих семантический анализатор ответа обучаемого. Она предназначалась для создания автоматизированных обучающих и контролирующих программ по различным дисциплинам и диагностическим тестам. Семантический анализатор основывался на семантической маске (интегральном образе возможных ответов обучаемого). В этой системе форма записи маски была в виде суперпозиции логических функций, аргументами которых являлись одноместные предикаты наличия или отсутствия в строке ответа обучаемого маскируемых дескрипторов. В маске в качестве аргументов записывались не значения предикатов, а сами дескрипторы, а значения предикатов наличия или отсутствия дескрипторов в строке ответа высчитывались непосредственно при разборе логического выражения. Для повышения скорости разбора логического выражения и минимизации оперативной и дисковой памяти форма записи логического выражения была нетрадиционной. Данная форма представляла собой некоторый синтез приемов записи полной скобочной и польской обратной записи.

Системы учебного назначения "Адонис", "Сценарий", "Hypermethod" предназначены для создания мультимедийных обучающих программ. В анализаторе свободно конструируемых ответов применялся эталонный метод и анализ по ключевым словам. Эталонный метод основан на сравнении ответа с заранее подготовленными эталонами. В зависимости от совпадения ответа с одним из эталонов выбирается управляющее воздействие. При анализе по ключевым словам определялось наличие или отсутствие перечисленных в списке дескрипторов в строке ответа. Анализ синонимов в каждой из этих систем реализован посредством оператора "ИЛИ". При анализе ответа пользователя на вопрос система накладывает на ответ заданную маску и определяет, удовлетворяет ли ответ такой маске или нет. Если ответ удовлетворяет маске, то он считается верным, и наоборот.

Система автоматической верстки обучающих программных средств "Виртуальный клон", разработанная в Тульском политехническом университете, предназначена для создания мультимедийных обучающих и контролирующих программ. Кроме визуальных компонентов (меню, списка, выборки и т. д.) в системе существует блок семантического анализа. Автор (учитель) задает семантическую маску путем перечисления дескрипторов (слов), назначения перечисленных дескрипторов в определенные рубрики, присваивание дескрипторам определенных семантических весов и определение связок между дескрипторами. Существует понятие жесткой и мягкой связи между дескрипторами. Жесткая связь определяет, что между дескрипторами, следующими в высказывании в определенном порядке, не должно существовать других слов. Мягкая связь определяет то, что дескрипторы должны находиться один после другого, независимо от того, есть ли между ними какие-либо другие слова или нет. Порядок анализа следующий: система берет первый дескриптор с наибольшим семантическим весом из наиболее важной рубрики, которую определяет преподаватель, отыскивает этот дескриптор в строке ответа, смотрит, существует ли жесткая или мягкая связь с другими дескрипторами, и ищет дескриптор, который находится в связи с этим. И так до тех пор, пока не будет найден дескриптор, который не имеет семантической связи. В этом случае берется дескриптор наиболее значимый по семантическому весу, в наиболее значимой рубрике. Результат определяется по суммарному семантическому весу найденных дескрипторов.

5.7.4. Системы искусственного интеллекта

Вопрос "может ли компьютер думать?" ничуть не более интересен,
чем вопрос "может ли подводная лодка плавать?"

Эдсгер Дейкстра

Мысль умирает первой.

Надежда

Системы искусственного интеллекта решают общую задачу семантического анализа естественного языка. Предназначены такие системы для ведения диалога с пользователем на естественном языке, семантического анализа текстов, получения знаний от пользователя и т. д.

Система семантического анализа компании Artificial Life, Inc. (**http://www.artificial-life.com**) выполняет семантический анализ языка на основе шаблонов. Эта система предназначена как для семантического анализа текстов, так и для анализа отдельных фраз, введенных пользователем. Система позволяет анализировать отдельные документы и выдавать краткий отчет, в котором отражается семантический смысл документа, или же вести диалог с пользователем на естественном языке. Диалог с пользователем ведется с помощью робота. Робот может отвечать на вопросы, сам задавать вопросы, что-то рассказывать пользователю и изменять тему разговора. Ядром системы является модуль ALife Smart Engine. Этот модуль содержит функции, реализующие алгоритмы семантического анализа. Alife Smart Engine работает исключительно на шаблонах. Шаблон представляет собой направленный граф, составленный из слов, и поэтому существует поддержка нескольких языков. Необходимо только написать в шаблонах слова на соответствующем языке.

Виталием Алексеевичем Тузовым [Тузов 1990] было предложено решение общей проблемы семантического анализа русского языка. На основе введенной пользователем фразы строится суперпозиция логических функций. Алгоритм анализа основан на поиске семантических связок слов в высказывании. Все семантические связки между словами хранятся в специальном словаре. Система сама определяет, какое слово какому подчинено, где подлежащее, сказуемое, наречие, определение и т. д. Полностью реализован анализ взаимодействия слов по синтаксическим признакам, предлогам, приставкам, суффиксам и т. д. На основе найденных связок слов строится суперпозиция логических функций. Недостатком данной системы является как необходимость наличия словаря слов и их форм, правил сочетания слов, так и постоянное их обновление.

5.8. Заключение

5.8.1. Выводы и рекомендации

Я знаю веселые сказки таинственных стран
Про черную деву, про страсть молодого вождя,
Но ты слишком долго вдыхала тяжелый туман,
Ты верить не хочешь во что-либо кроме дождя.
Николай Гумилев

Долгое время в системы программирования верили как в радикальное средство повышения производительности программирования, проектирования, тестирования и т. п. Ожидания оправдались лишь частично, резкого повышения производительности не произошло. Поэтому сейчас к системам относятся как к важным элементам, используемым в процессе разработки программных продуктов, но не "обожествляют" их.

Большинство инструментов и систем, применяемых сейчас разработчиками, имеет солидную историю, и они созданы давно и хорошо известными компаниями. Однако периодически появляются оригинальные идеи и инструменты. Например, упомянем интегрированную среду разработчика Eclipse (**http://www.ibm.com**), позиционируемую как среду "для всего вообще и одновременно для ничего конкретного". А слава хорошо продуманной среды Idea (**http://www.jetbrains.com/**) затмила аналогичные разработки гораздо более мощных компаний.

Тезисно сформулируем некоторые общие рекомендации, связанные с инструментарием разработчика.

- Следует использовать только те средства, в которых разработчики действительно будут нуждаться, и отказаться от ненужных и модных. Лучше определиться в этом как можно раньше, перед началом работы. Переход на новый инструмент в процессе работы отнимет большое количество времени.
- В основном следует использовать готовые инструменты, однако часть инструментов может быть разработана самостоятельно. Как правило, это небольшие инструменты, разработанные на скриптовых языках. Ряд компаний предлагает полуфабрикаты инструментов (например, части компиляторов), которые можно дополнить самостоятельно до готового инструмента.
- В команде разработчиков должно быть единое мнение относительно используемых средств. Например, все разработчики должны использовать компилятор определенной компании и, как правило, одну и ту же его версию.
- Некоторые инструменты профессиональные программисты должны использовать обязательно. В первую очередь, это относится к системам управления версиями файлов.
- Знакомство с новыми инструментами предполагает определенные временные затраты на обучение. Переходы на новый инструмент или на новую версию старого инструмента влекут за собой определенные риски. Например, новая версия компилятора может более строго следовать стандарту языка или содержать ошибки, отсутствующие ранее. Известно много случаев, когда коллективы разработчиков по этим причинам пользовались компиляторами более чем пятилетней давности.

Интересные выводы о развитии систем программирования можно сделать, наблюдая за статистикой проектов на сайте **http://sourceforge.net**, предоставляющем инфраструктуру для разработки свободного программного обеспечения. В табл. 5.3 приведены данные по количеству проектов для семи лидирующих направлений в области систем программирования на начало 2002 года и два года спустя. Видно, что лидерство за два года практически не поменялось. Кроме того, общая тенденция такова, что по всем направлениям количество проектов увеличилось примерно в 2 раза. Лидирующие классы систем привлекают наибольшее внимание разработчиков и одновременно

дают сигнал о том, что рынок недостаточно насыщен качественными и отвечающим своевременным потребностям пользователей системами.

Таблица 5.3. *Тенденции в развитии систем программирования*

Классы систем	2002	2004
Инструменты сборки	586	1338
Кодогенераторы	522	1201
Интерпретаторы	481	883
Компиляторы	393	752
Системы контроля версий	223	403
Отладчики	174	376
Брокеры объектов (Object Brokering)	201	357

Выбор инструментальной системы в самом общем случае может осуществляться по следующим соображениям. На рис. 5.14 по горизонтальной оси мы оцениваем сложность разрабатываемой программы, а по вертикальной — количество будущего программного кода. В самом простейшем случае нам потребуются инструментальные средства. Сложность программы ведет к необходимости использования интегрированной среды и мощного репозитория, а большой объем кода — к CASE-системам. В случае и сложности, и большого объема требуются мощные интегрированные окружения на базе языковых платформ.

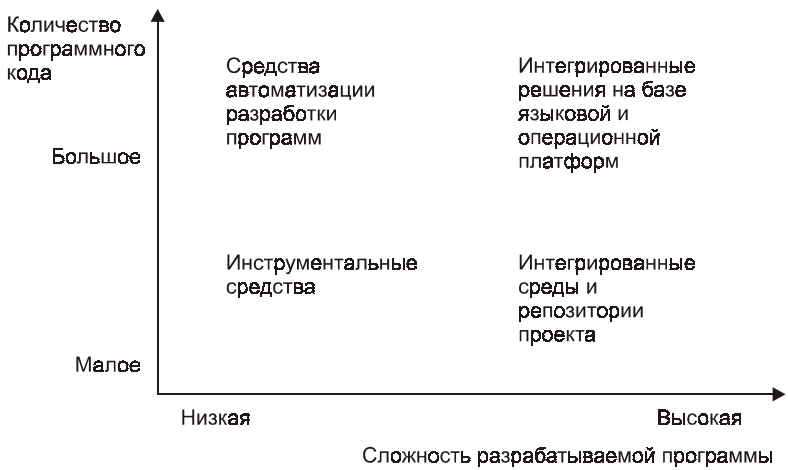


Рис. 5.14. Классы инструментальных систем

Как это не покажется странным, но и в системах программирования могут иметься небезопасные места, которыми могут воспользоваться злоумышленники. Например, в обыкновенном компиляторе при работе могут создаваться (а могут и не создаваться) временные файлы с определенными именами. Если перед началом работы компилятор не проследит, что они присутствуют, и не удалит их, злоумышленник может положить свой файл, который на завершающем этапе будет взят и окажется в результирующем бинарном файле.

Наконец, заметим, что в выбор инструментария очень часто вмешиваются политические соображения. Приведем типичный пример. В одной из крупных американских компаний в разгар ее судебного процесса против Microsoft запретили использование ряда программ, в том числе Microsoft Word, рекомендовав перейти на ее аналог — программу StarOffice.

5.8.2. Проблемы и перспективы развития

Обстоятельства меняются, принципы нет.
О. Бальзак

Характеризуя современное развитие систем программирования, можно отметить сложное взаимоотношение между дифференциацией и интеграцией инструментальных средств. Развитие и совершенствование отдельного инструментария вело к росту дифференциации, отделения и образования новых инструментов. Жесткая архитектура традиционных систем программирования часто оказывалась не в состоянии обеспечить их последующую интеграцию в гибкую согласованную систему.

Отечественные системы программирования и трансляторы характеризовались в большей степени исследовательским уровнем, чем промышленным качеством [Поттосин 1995]. Отсутствие реального рынка и конкуренции программных продуктов привело к тому, что даже широко использовавшиеся трансляторы не соответствовали международным требованиям на программный продукт.

Часто может оказаться, что достаточно быстро разработать инструментарий для собственных нужд может практически любая команда. Любопытную историю приводят в Интернете (**http://progstone.nm.ru**): "Одна команда пришла в волнение, когда они увидели инструменты, которые давали им все возможности для получения и просмотра перекрестных ссылок. Однако небольшие отличия между этими инструментами и набором скриптов, которые у них уже были и на написание которых понадобилось одно утро, заключались в следующем. Инструменты нельзя было модифицировать, они стоили 20 000 фунтов стерлингов (плюс 5000 за каждое рабочее место) и требовали несколько недель на установку и настройку".

Следует обратить внимание на необходимость создания принципиально новых инструментов, а не излишней шлифовке давно существующих. В ряде случаев шлифовка подобна улучшению молотка путем раскрашивания его ручки хохломской росписью.

Наконец, очевидное наблюдение заключается в том, что каждый программист естественно стремится подобрать удобные инструменты для себя, игнорируя широкие возможности универсальных средств.

Воздействие сети Интернет на развитие систем программирования также велико. Перспективными направлениями развития систем программирования в Интернете являются:

- проектирование и реализация инструментария, обеспечивающего кроссплатформенную разработку приложений на языках Java, C и C++ в средах ОС Unix и Windows;
- разработка инструментария для эффективного написания программ, поддерживающих распределенные вычисления;
- разработка высокопроизводительных систем программирования;
- разработка средств отладки интернет-приложений;
- разработка инструментария с высоким уровнем абстракции для проектирования интерфейсов.

Упомянем и возможности, которые предлагает Интернет для доставки систем программирования до пользователя.

Пусть нам потребовался высокопроизводительный оптимизирующий компилятор с языка программирования C++ для SPARC-архитектуры. Вот два его наиболее удачных варианта:

- профессиональный коммерческий C++ компилятор компании Sun Microsystems, Inc. (**http://access1.sun.com/**);
- свободно распространяемый под General Public License (GPL) лицензией C++ (**http://gcc.gnu.org/**).

Интернет содержит достаточное количество ресурсов и для образовательных целей. Конечно, особый интерес для учебных заведений представляют свободно распространяемые программы. Многие учебные заведения используют язык Pascal для первоначального обучения программированию. В Интернете можно найти свободно распространяемые компиляторы с языка Pascal:

- Free Pascal (**http://www.freepascal.org/**);
- GNU Pascal (**http http://www.gnu-pascal.de/gpc/h-index.html**).

Среди других доступны компиляторы с языков C, C++, Eiffel, Lisp, Clos, FORTRAN и многие другие.

5.8.3. Рекомендации по литературе

Читайте книги — некоторые из них специально для этого написаны.
Михаил Генин

До сих пор не было издано современных книг, дающих хороший обзор и сравнительный анализ существующих систем программирования. Этому имеется очень простое объяснение. Программисты находятся в условиях жесткого рыночного давления со стороны производителей платформ. Бескорыстный поиск истины в такой ситуации выглядит просто несколько неестественным. С ностальгией приходится вспоминать времена 30-летней давности, когда появлялись работы, в которых на академическом уровне проводилось сравнение, например, компиляторов, без указания компании-разработчика и даже без номеров версий.

Конечно, такая ситуация не является большой бедой. Гораздо важнее охарактеризовать совокупность задач, решаемых системами программирования и изложить методы их решения. Можно отметить прекрасные работы, отражающие различные теоретические и практические аспекты наиболее часто используемых инструментов. Например, для трансляторов с языков программирования это следующие книги:

- "Теория синтаксического анализа, перевода и компиляции" [Ахо, Ульман 1978] и "Компиляторы: принципы, технологии и инструменты" [Ахо, Сети, Ульман 2001], содержащие теоретические основы конструирования компиляторов;
- "Конструирование компиляторов для цифровых вычислительных машин" [Грис 1975], в которой рассматриваются практические вопросы реализации компиляторов;
- "Как Паскаль и Оберон попадают на "Самсон", или искусство создания трансляторов" [Кожохарь, Евстюнин, Терехов, Уфнаровский 1992], в которой приведены примеры практически готовых трансляторов для отечественной ЭВМ "Самсон";
- "Языки и методы программирования в системе "Эльбрус" [Сафонов 1989], в которой приведены практические сведения об использовании систем программирования, реализованных на вычислительных комплексах "Эльбрус";
- "Текущее состояние российских исследований и разработок в области трансляции" [Поттосин 1995]. Эта работа дает представление об отечественных разработках последних лет и содержит ссылки на российские коллективы и центры, занимающиеся разработкой систем программирования.

Обратим внимание на то, что программисту всегда следует ориентироваться на два основных документа, сопровождающих каждый программный продукт: руководство пользователя и справочное руководство.

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Программные средства и окружения" (Programming Tools and Environment). Основные разделы этой темы:
  - среды программирования;
  - анализ требований и средства моделирования проекта;
  - инструментальные средства тестирования;
  - инструментальные средства управления конфигурацией;
  - механизмы интеграции инструментальных средств.
- Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет следующие дисциплины специализации и разделы в них:
  - трансляторы;
  - базы данных и СУБД;
  - системы искусственного интеллекта.

5.8.4. Вопросы и задания

1. Что традиционно входит в состав системы программирования?
2. Проведите классификацию систем программирования, отражая их функциональную ориентацию в технологическом процессе.
3. Проанализируйте и оцените набор инструментов, поддерживающих классические технологические процессы.
4. Опишите отличия между компилятором и интерпретатором.
5. Какие аппаратные средства и средства операционных систем используют отладчики для отладки программ?
6. В какой информации от компилятора и компоновщика (редактора связей) нуждается отладчик?
7. Приведите примеры инструментов работы с текстом.
8. Каковы характерные особенности систем программирования, созданных в рамках общинных проектов (например, проекта GNU)?
9. В чем заключаются основные особенности разработки программного обеспечения для сети Интернет?
10. Почему одной из тенденций развития современных систем программирования является включение в их состав синтаксически-ориентированных редакторов?

11. Что такое интерфейс прикладного программиста, и какие задачи решаются в прикладных программах с его помощью?
12. В каких случаях в настоящее время продолжают использоваться средства разработки на основе makefile?
13. Каким образом электронные библиотеки и инструментарий Интернет можно использовать в профессиональном поиске информации?
14. Перечислите достоинства и недостатки используемых вами систем программирования.
15. Проведите сравнительный анализ инструментальных сред программирования, CASE-средств и интегрированных сред.
16. Какова роль репозитория в современных инструментальных системах?
17. Почему системы управления версиями файлов следует использовать не только при коллективной, но и при индивидуальной разработке?
18. Какими могут быть общие требования к системам поддержки работы виртуальных групп?
19. В чем заключаются основные проблемы, возникающие при создании систем искусственного интеллекта.
20. Выберите и обоснуйте набор инструментальных систем для поддержки разработки ряда программных продуктов.

ЛИТЕРАТУРА

Можно определить достоинство народа по количеству книг,
которые он поглощает.
Э. Лабуле

1. *Ахо, Сети, Ульман 2001* — Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. — М.: Издательский дом "Вильямс", 2001.
2. *Ахо, Ульман 1978* — Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х томах. — М.: Мир, 1978.
3. *Бульонков, Бабурин 2000* — Бульонков М. А., Бабурин Д. Е. HYPERCODE — открытая система визуализации программ // Автоматизированный реинжиниринг программ. — СПб.: Изд-во СПбГУ, 2000.
4. *Вендров 1998* — Вендров А. М. CASE-технологии. Современные методы и средства проектирования информационных систем. — М.: Финансы и статистика, 1995.
5. *Вооглайд, Серебряков 1989* — Вооглайд А. О., Серебряков В. А. Применение систем автоматизации построения трансляторов // Прикладная информатика. Выпуск 15. — М.: Финансы и статистика, 1989.
6. *Гаврилова, Хорошевский 2001* — Гаврилова Т. А., Хорошевский В. Ф. Базы знаний интеллектуальных систем. — СПб.: Питер, 2001.
7. *Грис 1975* — Грис Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.

8. *Дейт 2000* — Дейт К. Введение в системы баз данных, 6-е изд. — К.; М.; СПб.: Издательский дом "Вильямс", 2000.

9. *Жоголев 1996* — Жоголев Е. А. Лекции по технологии программирования. — М.: МГУ, факультет ВМиК, рукопись, 1996.

10. *Искусственный 1990* — Искусственный интеллект: В 3-х кн. Справочник. — М.: Радио и связь, 1990.

11. *Калянов 1996* — Калянов Г. Н. CASE структурный системный анализ (автоматизация и применение). — М.: ЛОРИ, 1996.

12. *Когаловский 1999* — Когаловский М. Р. Научные коллекции информационных ресурсов в электронных библиотеках // Электронные библиотеки: Перспективные методы и технологии, электронные коллекции: Труды Первой Всероссийской научной конференции. С.-Петербург, 19—22 октября 1999 г. — СПб.: Изд-во СПбГУ, 1999, с. 16—31.

13. *Кожохарь, Евстюнин, Терехов, Уфнаровский 1992* — Кожохарь С. К., Евстюнин М. В., Терехов А. Н., Уфнаровский В. А. Как Паскаль и Оберон попадают на "Самсон", или искусство создания трансляторов. — Кишинев: "Штиинца", 1992.

14. *Краснослободцев, Смирнов, Лиходедов 1998* — Краснослободцев В. Я., Смирнов А. Б., Лиходедов Н. П. Инновационный инжиниринг. Практикум. Учебное пособие. — СПб.: Государственный технический университет, 1998.

15. *Мартин 1982* — Мартин Д. Организация баз данных в вычислительных системах. — М.: Финансы и статистика, 1982.

16. *Мартыненко1997* — Мартыненко Б.К. Синтаксически управляемая обработка данных — СПб.: Изд-во СПбГУ, 1997.

17. *Некрестьянов, Пантелеева 2002* — И. Некрестьянов, Н. Пантелеева. Системы текстового поиска для Веб // Программирование, 28(4): 207-225, 2002.

18. *Нильсон 1985* — Нильсон Н. Принципы искусственного интеллекта. — М.: Радио и связь, 1985.

19. *Паронджанов 1999* — Паронджанов В. Д. Как улучшить работу ума. — М.: Радио и связь, 1999.

20. *Попов 1987* — Попов Э. В. Экспертные системы. — М.: Наука, 1987.

21. *Поттосин 1995* — Поттосин И. В. Текущее состояние российских исследований и разработок в области трансляции. — Новосибирск: Изд-во ИСИ СО РАН, 1995.

22. *Поттосин 1997* — Поттосин И. В. Программистская инженерия: содержание, мнения и тенденции // Программирование. — 1997. — № 4. — С. 26—37.

23. *Сафонов 1989* — Сафонов В. О. Языки и методы программирования в системе "Эльбрус". — М.: Наука, 1989.

24. *Смирнов, Керов, Дерюшев 1998* — Смирнов В. М., Керов Л. А., Дерюшев В. А. Создание Windows и Internet-приложений в виде виртуальных книг. — СПб.: ЭЛБИ, 1998.

25. *Сорокин 1989* — Сорокин В. В. Многоуровневая архитектура технологического программного окружения. — Новосибирск: Изд-во НФ ИТМ и ВТ АН СССР, 1989.

26. *Тузов 1990* — Тузов В. А. Языки представления знаний. — Л.: Изд-во ЛГУ. — 1990.

27. *Цейтин 1990* — Цейтин Г. С. На пути к сборочному программированию // Программирование. — 1990. — № 1. — С. 78—92.

28. *Фаулер 2003* — Фаулер М. Рефакторинг: улучшение существующего кода. — СПб.: Символ-Плюс, 2003.

29. *Фаулер, Скотт 1999* — Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. — М.: Мир, 1999.

30. *Холчева 1999* — Холчева А. В. Система для создания и поддержки персональных электронных библиотек // Интернет и современное общество: Тезисы Второй всероссийской научно-методической конференции. Санкт-Петербург, 29 ноября — 3 декабря 1999 г. — СПб.: Изд-во СПбГУ, 1999, с. 95—96.

31. *Aho, Sethi, Ullman 1986* — Aho A., Sethi R., Ullman J. Compilers: Principles, Techniques, and Tools. Springer Verlag, 1986.

32. *Ambras, O'Day 1987* — J. Ambras, V. O'Day. Microscope: A program analysis system // Proc. 20th Hawaii Inter. Conf. Syst. Sci., 1987, pp. 460—468.

33. *Badger, Weiser 1988* — Badger L., W. M. Minimizing. Communications for Synchronizing Parallel Data-flow Programs // Proc. International Conference for Parallel Processing. 1988.

34. *Ball, Horwitz 1993* — Ball T., Horwitz S. Slicing Programs with Arbitrary Control-flow. Proceeding First International Workshop Automated and Algorithmic Debugging // LNCS. 1993. V. 749, pp. 206—222.

35. *Broder 2000* — Broder A.Z., Kumar S.R., Maghoul F., Raghavan P., Rajagopalan S., Stata R., Tomkins A., Wiener J. L. Graph structure in the web. Proc. of the WWW9, pp. 309-320, 2000.

36. *Chen, Nishimoto, Ramamoorthy 1990* — Y. Chen, M. Y. Nishimoto, C. V. Ramamoorthy. The C information abstraction system. IEEE Trans. Software Eng., vol. 16, no. 3, pp. 325—334, Mar. 1990.

37. *Codd, Codd, Salley 1993* — Codd E. F., Codd S. B., Salley C. T. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. E. F. Codd & Associates, 1993.

38. *Grass 1992* — J. E. Grass. Object-Oriented Design Archaeology with CIA++. Comput. Syst. J. USENIX Assn., vol. 5, no. 1, pp. 5—67, Winter 1992.

39. *Gruber 1993* — T. R. Gruber. A translation approach to portable ontologies. Knowledge Acquisition 5(2), 1993, pp. 199—220.

40. *ISO/IEC 14102:1995* — ISO/IEC 14102:1995 Information Technology — Guideline for the evaluation and selection of CASE tools.

41. *Jahnke, Heitbreder 1998* — J. H. Jahnke, M. Heitbreder. Design recovery of legacy database applications based on possibilistic reasoning. Proceedings of 7th IEEE Intl. Conf. Of Fuzzy Systems (FUZZ'98). Anchorage, USA. IEEE Computer Society Press, May 1998.

42. *Jansen, Sprink, Saracevic 2000* — B. J. Jansen, A. Sprink, T. Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. Information Processing and Management, 36(2), 2000, pp. 207—227.

43. *Kozaczynsky, Hing, Engberts 1992* — W. Kozaczynsky, J. Hing, A. Engberts. Program concept recognition and transformation. IEEE Trans. Software Eng., vol. 18, no. 12, pp. 1065—1075, Dec., 1992.

44. *Linton 1984* — M. A. Linton. Implementing relational views of programs // Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp., Practical Software Devel. Envir., May 1984.

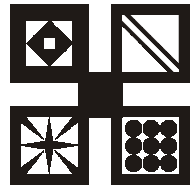
45. *Mahalingam, Huhns 1997* — K. Mahalingam, M. N. Huhns. A Tool for Organizing Web Information // Computer, June 1997, pp. 80—83.

46. *Muchnick 1997* — S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, 1997.

47. *Niere, Wadsack, Zundorf 2000* — J. Niere, J. Wadsack, A. Zundorf. Recovering UML Diagrams from Java code using patterns. University of Paderborn, Department of mathematics and computer science. 2000.

48. *Sharon, Bell 1995* — D. Sharon, R. Bell. Tools that bind: creating integrated environments // IEEE Software, March 1995, pp. 76—85.

49. *Sommerville 1992* — I. Sommerville. Software Engineering. — Addison-Wesley Publishing Company, 1992.



Глава 6

Архитектурная платформа

Вся эта глава посвящена компьютерным архитектурам. Нам будут интересны три основных группы вопросов.

- Основные архитектуры набора команд: классические, современные и нестандартные.
- Организация вычислительной системы: процессор, память, шина, функциональное описание.
- Параллельные и распределенные архитектуры, в том числе сети ЭВМ как частный случай распределенной архитектуры.

Один из разделов посвящен обзору одной из самых элегантных архитектур — SPARC-архитектуры.

6.1. Основы архитектуры ЭВМ

6.1.1. Основные понятия и определения

Архитектура ЭВМ (компьютера) — это совокупность архитектуры набора команд и организации вычислительной системы.

Архитектура набора команд (система команд) — граница между аппаратурой и программным обеспечением, представляющая собой ту часть системы, которая видна программисту или разработчику компиляторов.

Заметим, что архитектура компьютера определяет концептуальную структуру и функциональное поведение, но не отражает такие проблемы, как передача потоков данных внутри процессора, конструктивные особенности логических схем и специфика технологии производства.

Организация вычислительной системы — функциональные устройства и связь их некоторым образом в единую структуру. Простейшая вычислительная система может включать процессор, память, периферийные устройства и общую шину (рис. 6.1).

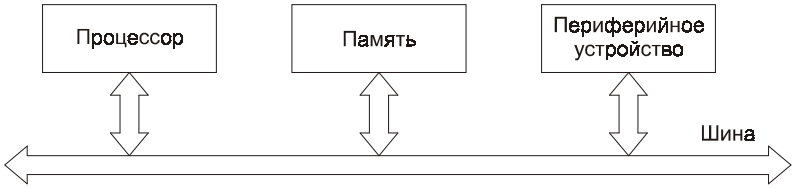


Рис. 6.1. Система с общей шиной

6.1.2. История и эволюция компьютерных архитектур

6.1.2.1. Поколения компьютерных архитектур

Вначале было слово, но это не было фиксированное число битов.
Р. С. Бартон

Рассматривать все множество архитектур можно с точки зрения истории по поколениям. Практически каждое поколение определяется новой элементной базой.

- Нулевое поколение. Появление первых реальных компьютеров. Конец 30-х годов XX века.
- Первое поколение. Компьютеры на вакуумных лампах. Появились в конце 40-х годов XX века.
- Второе поколение. Компьютеры на транзисторах. Появились в конце 50-х годов XX века.
- Третье поколение. Компьютеры на интегральных схемах. Появились в середине 60-х годов XX века.
- Четвертое поколение. Компьютеры на микропроцессорах. Появились в 70-х годах XX века.
- Пятое поколение. Интеллектуальные компьютеры. В качестве унифицирующего звена таких ЭВМ рассматривалось логическое программирование. Появились в начале 80-х годов XX века.

> ***О соотношении стоимости аппаратуры и рабочей силы***
>
> Обратим внимание на то, что исторически можно проследить три периода в соотношении стоимости аппаратуры и стоимости рабочей силы. Первый период (1945—1970 гг.) — дорогая аппаратура, дешевая рабочая сила. Второй период (1970—1985 гг.) — дешевая аппаратура, дорогая рабочая сила. И, наконец, третий (1981—...) — очень дешевая аппаратура, очень дорогая рабочая сила, численность персонала возрастает.

Любопытен тот факт, что в 1943 году Томас Уотсон (Thomas Watson), директор компании IBM, считал, что "на мировом рынке есть спрос только на пять штук компьютеров".

6.1.2.2. Некоторые важные даты

Приведем некоторые даты, связанные с появлением первых компьютеров.

- 1938 г. — Конрад Цузе (Conrad Zuse) завершил построение электромеханического двоичного компьютера Z1.
- 1939 г. — Джон Атанасофф (John Atanasoff) со своим студентом Клиффордом Берри (Clifford Berry) построил прототип электронного цифрового компьютера с использованием двоичной арифметики.
- 1943 г. — начал работать британский компьютер "Колосс". В его разработке принимал участие Алан Тьюринг (Alan Turing).
- 1944 г. — в Гарвардском университете 7 августа открыта построенная Говардом Айкеном (Howard Aiken) машина Howard Mark (она также известна как IBM ASCC — Automatic Sequence Controlled Calculator).
- 1945 г. — в школе электроинженерии при Пенсильванском университете Преспером Экертом (Presper Eckert) и Джоном Моучли (John Mauchly) построена машина ENIAC (Electronic Numerical Integrator and Computer). Этот компьютер был самым большим компьютером в истории — он весил 30 тонн.

Начиная с конца 40-х годов XX века, стали появляться первые компьютеры на вакуумных лампах. 1948 год можно считать годом появления электронных средств вычислительной техники и в СССР, и в США. Вот еще несколько дат, связанных с начальными этапами компьютерной архитектуры.

- В 1947—1948 гг. инженер Сергей Алексеевич Лебедев в Феофании под Киевом создал первый ламповый вычислительный блок — прообраз компьютеров.
- 1949 г. — под руководством Джея Форрестера (Jay Forrester) в Массачусетском технологическом институте был разработан первый компьютер для решения задач реального времени Whirlwind. Он содержал около 5000 вакуумных ламп.
- 1952 г. — первый компьютер Univac (Universal Automatic Computer) установлен в американском агентстве по переписи населения.
- 1957 г. — один из первых компьютеров на транзисторах Atlas Guidance Computer, выпущенный фирмой Burroughs, был применен при управлении запуском ракеты.
- 1959 г. — на выставке в Париже продемонстрирован первый японский коммерческий компьютер на транзисторах NEAC 2201 компании NEC.
- 1960 г. — компания DEC объявила о выпуске PDP-1 — первого коммерческого компьютера с монитором и клавиатурным вводом.
- 1964 г. — компания IBM объявила о производстве серии компьютеров System/360 и заняла ведущую позицию в выпуске компьютеров общего

назначения на многие годы. Кстати, долгое время велись многочисленные споры о том, как расшифровать число 360 в названии этой серии. Одни специалисты утверждали, что это "3-е поколение — 60-е годы (XX века)", а другие — "возможность использования в любом направлении — от 0 до 360 градусов".

6.1.2.3. История советской и российской вычислительной техники

В истории советской и российской вычислительной техники можно выделить четыре основных периода.

- Первый период продолжался с 1948 года до начала 70-х годов XX века. Он связан с такими именами, как С. А. Лебедев, М. В. Келдыш, М. А. Лаврентьев, А. А. Дородницын, Б. А. Бабаян, В. С. Бурцев, В. К. Левин, Г. Г. Рябов, И. С. Брук, Б. И. Карцев, Ю. Я. Базилевский и многими другими учеными и инженерами. Основные машинные разработки — М-1, М-2, Стрела, М-20, БЭСМ, БЭСМ-2, Раздан, Урал, Мир, Сетунь, Наири.
- Второй период продолжался с начала 70-х до начала 90-х годов XX века. Он характеризуется двумя тенденциями.
  - Частичное прекращение отечественных разработок и принятие решения о копировании американской техники. Именно тогда началось отечественное серийное производство ЭВМ.
  - Продолжение оригинальных отечественных разработок в некоторых научных институтах. Одним из них был Московский институт точной механики и вычислительной техники (ИТМ и ВТ), в котором было создано семейство отечественных суперкомпьютеров Эльбрус.
- Третий период продолжался с начала 1990-го года по 2000-й год и характеризовался существенным прекращением выпуска отечественной вычислительной техники. Заводы перестали выпускать "наши" ЭВМ, поскольку они по причине низкого качества не могли конкурировать с западными ЭВМ.
- Четвертый период начался в 2000 году и характеризуется возрождением отечественных разработок в области архитектуры ЭВМ.

Приведем некоторые даты создания отечественных ЭВМ (линия Лебедева и его последователей).

- МЭСМ:
  - 1951—1952 гг. — создание прототипа.
- БЭСМ:
  - 1953 г. — создание прототипа.

- М-40:
  - 1957—1959 гг. — первый компьютер Эльбрус на вакуумных лампах.
- БЭСМ-2:
  - 1958 г. — создание прототипа;
  - 1959 г. — начало серийного выпуска.
- М-20:
  - 1958 г. — создание прототипа;
  - 1959 г. — начало серийного выпуска. Для своего времени М-20 была одной из самых быстродействующих машин первого поколения в мире.
- БЭСМ-3М/БЭСМ-4:
  - 1962—1963 гг. — создание прототипа;
  - 1964 г. — начало серийного выпуска.
- 5Э92б:
  - 1964—1969 гг. — компьютер на дискретных транзисторах. Использовался в первой Российской противоракетной системе обороны Москвы.
- БЭСМ-6:
  - 1964—1967 гг. — создание прототипа;
  - 1967 г. — начало серийного выпуска.
- Эльбрус-1:
  - 1973—1979 гг. — компьютер на основе суперскалярного RISC-процессора.
- Эльбрус-2:
  - 1977—1984 гг. — 10-процессорный суперкомпьютер. Использовался в Российской противоракетной системе.
- Эльбрус-3:
  - 1985—1993 гг. — 16-процессорный компьютер с большой локальной оперативной памятью для каждого процессора (16 Мбайт) и глобальной, общей для всех процессоров памятью (2 Гбайт).
- Эльбрус-90 Микро:
  - 1996 г. создан опытный образец компьютера.
- Эльбрус-2000 (Э2К):
  - 1994—... разработка архитектуры postRISC с явным параллелизмом и системой двоичной компиляции. К 1999 г. архитектура была описана на языке Verilog.

6.1.2.4. Эволюция понятия компьютерная архитектура

Понятие компьютерной архитектуры претерпевало изменения с момента своего появления. Особенно это заметно на примере основного материала, изучаемого в лекционных курсах по архитектуре.

- 50—60-е годы XX века — компьютерная арифметика.
- 70-е — середина 80-х годов XX века — проектирование набора команд, особенно удобных с точки зрения компиляторов.
- 1990-е годы XX века — проектирование организации вычислительных систем, включая процессор, память, системы ввода-вывода, а также многопроцессорные системы.

6.1.3. Классификация вычислительных систем

6.1.3.1. Функциональная классификация компьютеров

Суперкомпьютер — это устройство, сводящее проблему вычислений к проблеме ввода-вывода.

Кен Батчер

С точки зрения *функциональной* классификации компьютеров, можно выделить:

- *массивно-параллельные системы* — однородные вычислительные узлы со скоростными специализированными каналами связи;
- *кластерные архитектуры* — дешевый вариант массивно-параллельных систем;
- *суперкомпьютеры* — высокоскоростные вычислители;
- *мэйнфреймы* — большие универсальные компьютеры;
- *мини-компьютеры* — компьютеры, появившиеся как альтернатива мэйнфреймам и имеющие гораздо меньшие по сравнению с ними размеры. Эпоха мини-компьютеров приходится на 80-е годы XX века;
- *рабочие станции* — высококачественные компьютеры для рабочих мест пользователей. Они вытеснили мини-компьютеры, завершив их эпоху. В модели вычислений "клиент-сервер" принято выделять разновидность данного класса — *рабочие станции-серверы*, выполняющие часть клиентской работы (и поэтому, обычно, имеющие более мощную конфигурацию). В этой же модели выделяют еще одну разновидность — *бездисковые рабочие станции*, используемые, как правило, в качестве клиентов;
- *терминалы* — оконечные устройства, не имеющие собственного процессора общего назначения. Такие устройства фактически не являются вычислителями, а выполняют только операции ввода-вывода;
- *персональные компьютеры* — универсальные компьютеры, рассчитанные на одного пользователя и управляемые одним человеком. В модели вычислений "клиент-сервер" принято выделять разновидность данного класса — *персональные компьютеры-серверы*;

- *переносимые компьютеры* — лэптопы, ноутбуки;
- *карманные компьютеры* — палмтопы;
- *встроенные компьютеры* — компьютеры для специальных целей.

На рис. 6.2 функциональные классы компьютеров представлены на координатной плоскости "стоимость — мощность". Рисунок отражает сегменты вычислительных систем на конец 90-х годов XX века.

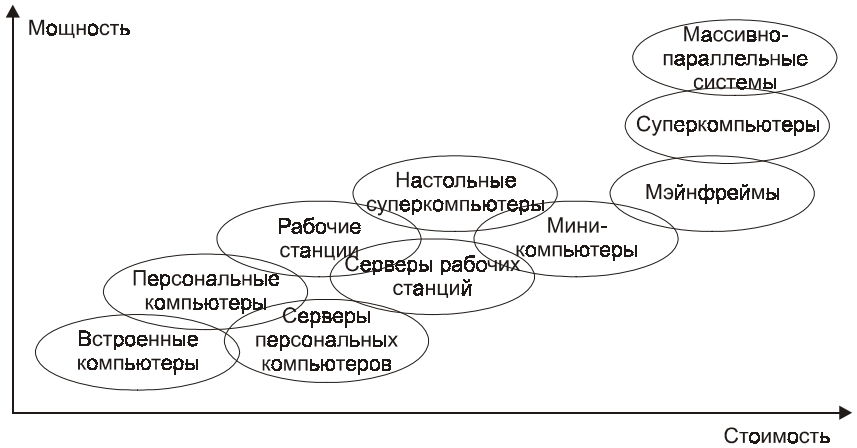


Рис. 6.2. Функциональные классы компьютеров (конец 90-х годов XX века)

Существует международный сертификационный стандарт, регламентирующий принципы классификации персональных компьютеров и оговаривающий минимальные и рекомендуемые требования к каждой категории. Стандарт устанавливает следующие категории персональных компьютеров:

- *массовый персональный компьютер* (Consumer PC). В эту категорию попадают большинство персональных компьютеров, не имеющих специфических повышенных или пониженных требований, которые могут перевести компьютеры в другую категорию;
- *деловой персональный компьютер* (Office PC). Для этой категории минимизированы требования к средствам воспроизведения графики, а к средствам работы со звуковыми данными требования вообще не предъявляются;
- *портативный персональный компьютер* (Mobile PC). Обязательным является наличие средств связи для удаленного доступа;
- *персональная рабочая станция* (Workstation PC). Для этой категории повышены требования к устройствам хранения данных;
- *развлекательный персональный компьютер* (Entertainment PC). Для этой категории повышены требования к средствам воспроизведения графики и звука.

6.1.3.2. Классификация по степени централизации

Еще одна классификация аппаратных систем связана со степенью централизации. Она определяет, насколько сильной или слабой связностью обладают аппаратные системы (рис. 6.3). Эта характеристика нам будет особенно важна при разработке программного обеспечения (это мы увидим уже в *гл. 7*), поскольку алгоритмы программирования для аппаратных систем разной степени связности существенно отличаются.

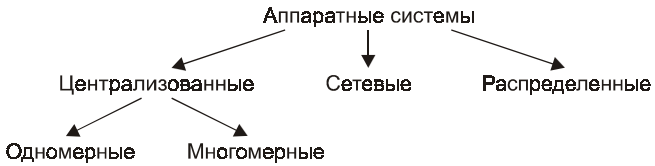


Рис. 6.3. Классификация аппаратных систем по степени централизации

6.1.3.3. Классификация архитектур набора команд

Поскольку архитектура набора команд в большой степени характеризует компьютерную архитектуру, приведем схему классификации, базирующуюся на наборе команд (рис. 6.4). Так как англоязычные аббревиатуры (и некоторые названия) архитектур очень широко распространены в литературе, мы специально приведем их рядом с русскоязычными названиями.

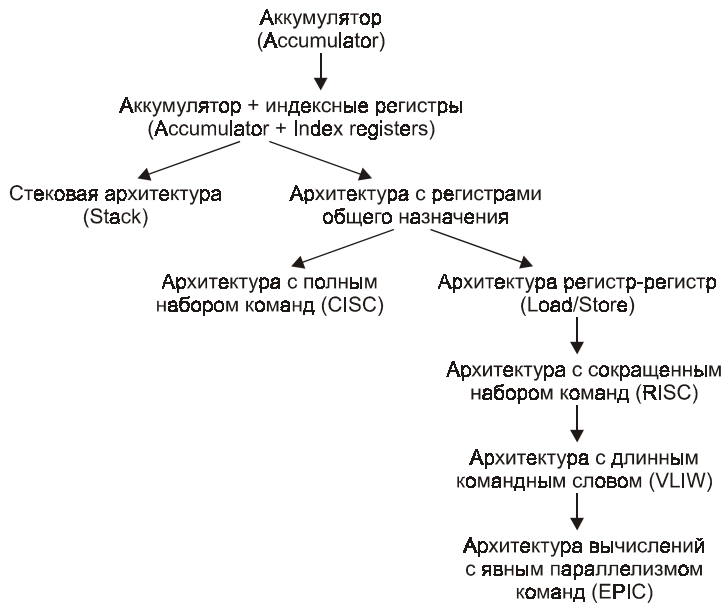


Рис. 6.4. Эволюция архитектуры набора команд

Обратим внимание на то, что архитектуры набора команд на этой схеме обозначают идеализированные концепции. Реальные процессоры иногда бывает достаточно сложно отнести к одной из них.

6.1.3.4. Классификация Флинна

Классификация Флинна (Michael Flynn) была предложена в конце 60-х годов XX века и фактически является первым вариантом классификации вычислительных систем. Она основана на идее описания параллелизма по двум измерениям: параллелизм потока команд и параллелизм потока данных. Согласно этой классификации все системы могут быть разделены на четыре класса.

- *Одиночный поток команд и одиночный поток данных* (single instruction stream and single data stream — SISD). Данный класс включает компьютеры, основанные на классической фон Неймановской архитектуре *(см. разд. 6.2.1.1)*. Фактически с 40-х до 70-х годов XX века этот класс был единственным.
- *Одиночный поток команд и множественный поток данных* (single instruction stream and multiple data stream — SIMD). Ускорение в вычислениях достигается за счет разделения всей вычислительной работы на последовательность некоторых специализированных этапов и организации передачи обрабатываемых данных от текущего этапа к следующему.

  О классе SIMD

  Ряд авторов считают, что не существует реальной вычислительной системы, построенной на данном принципе. Другие авторы полагают, что к этому классу относятся конвейерные или магистральные системы. Мы будем считать далее, что данный класс пуст.

- *Множественный поток команд и одиночный поток данных* (multiple instruction stream and single data stream — MISD). Системы этого класса также называют матричными процессорами или ассоциативными устройствами. Данные вычислительные системы характеризуются наличием нескольких идентичных процессорных элементов, работающих под управлением одного устройства управления. На каждом такте устройство управления предписывает исполнение одной и той же команды на всех процессорных элементах.
- *Множественный поток команд и множественный поток данных* (multiple instruction stream and multiple data stream — MIMD). Вычислительные системы представляют собой многопроцессорный вычислительный комплекс, каждый процессор которого имеет собственное устройство управления.

6.1.3.5. Другие классификации

Системы параллельной обработки данных имеют свои специфические классификации, которые будут описаны в *разд. 6.5*.

Классификации компьютерных сетей будут рассмотрены в *разд. 6.6*.

6.2. Основные архитектуры набора команд

6.2.1. Классические архитектуры

Компьютеры являются очень простыми устройствами. Они гораздо проще, чем люди, которые их создали, и студенты, которые их изучают.

Ричард Пол

В этом разделе речь пойдет о тех архитектурах, с которых начиналась история. В первую очередь это фон Неймановская архитектура. Далее следуют аккумулятор, стековая и регистр-регистр архитектуры.

6.2.1.1. Фон Неймановская архитектура

Архитектура большей части современных вычислительных систем, по существу, близка архитектуре, предложенной Джоном (Яношем) фон Нейманом (John von Neumann) в 40-х годах XX века. Таким вычислительным системам присущи следующие характеристики [Майерс 1985]:

- единственная последовательно адресуемая память. Программа и данные хранятся в одной памяти, адреса областей которой составляют последовательность;
- память является линейной и одномерной;
- отсутствует явное различие между командами и данными. Их идентифицируют неявным способом при выполнении операций;
- назначение данных не является их неотъемлемой составной частью. Оно определяется логикой программы.

Цикл исполнения команд на машине, построенной на основе фон Неймановской архитектуры, приведен в листинге 6.1.

Листинг 6.1. Цикл исполнения команд для фон Неймановской архитектуры

```
program_counter = 0;          /* инициализация программного счетчика */
do {
    instruction = memory[program_counter++];  /* извлечение команды */
    decode(instruction);      /* декодирование команды */
    fetch(operands);          /* извлечение операндов */
    execute;                  /* исполнение команды */
    store(results);           /* запоминание результатов */
} while ( instruction != halt);
```

6.2.1.2. Архитектура аккумулятор

Архитектура *аккумулятор* (Accumulator architecture) очень проста и наиболее близка к архитектуре простейшего калькулятора. Процессор имеет единственный регистр — аккумулятор, давший название архитектуре (рис. 6.5). Его содержимое комбинируется в арифметико-логическом устройстве (АЛУ) процессора с единственным операндом. Результат операции также помещается в аккумулятор. Существуют две команды для загрузки значения из памяти в аккумулятор и выгрузки его из аккумулятора в память. Данная архитектура использует адрес основной памяти в качестве единственного операнда команды.

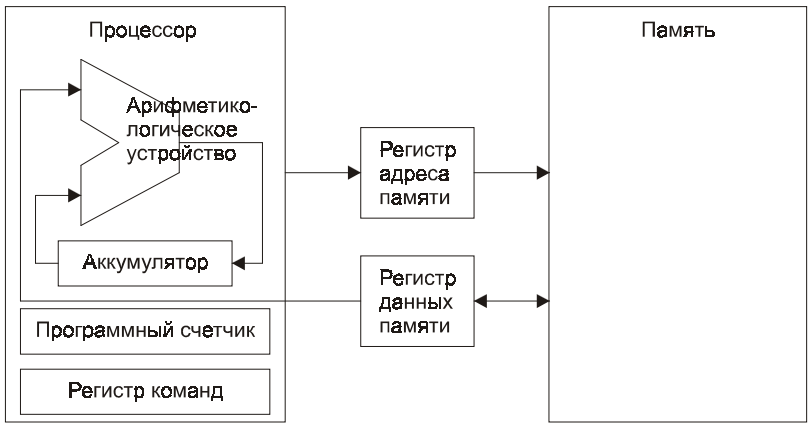


Рис. 6.5. Архитектура аккумулятор

Так в данной архитектуре выглядит последовательность машинных команд, которая может использоваться для сложения двух чисел:

```
load   1000 /* загрузить в аккумулятор значение по адресу 1000 */
add    1002 /* сложить содержимое аккумулятора со значением
               по адресу 1002 */
store  1003 /* результат, содержащийся в аккумуляторе, поместить
               по адресу 1003 */
```

6.2.1.3. Архитектура регистр-регистр

Архитектура *регистр-регистр* (Load/Store architecture) характеризуется наличием высокоскоростной регистровой памяти, расположенной на процессоре (рис. 6.6). Эта архитектура близка к стековой, но операнды могут быть взяты из любого регистра высокоскоростной памяти. С помощью специальных команд можно производить обмен данными между регистровым файлом (регистровой памятью) и основной памятью. Арифметические и логические команды в качестве операндов используют три регистра: два из них являются источниками, а в третий помещается результат.

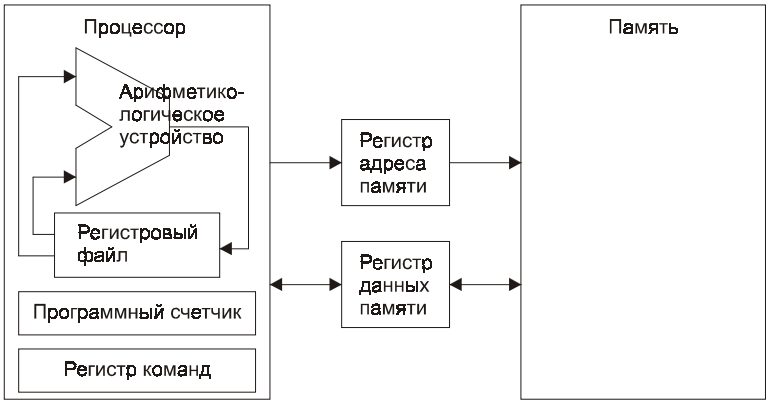


Рис. 6.6. Архитектура регистр-регистр

Для сложения двух чисел в данной архитектуре достаточно одной машинной команды:

```
add     reg_src_1, reg_src_2, reg_dest /* сложить содержимое
        регистра reg_src_1 с содержимым регистра reg_src_2
        и результат поместить в регистр reg_dest */
```

6.2.1.4. Стековая архитектура

Арифметические команды *стековой* архитектуры (Stack architecture) не имеют операндов. Оба операнда берутся с вершины стека, и результат также помещается на вершину стека (рис. 6.7). Существуют две команды для загрузки значения из памяти на вершину стека и выгрузки значения на вершине стека в память. Эти команды использует адрес основной памяти в качестве единственного операнда команды.

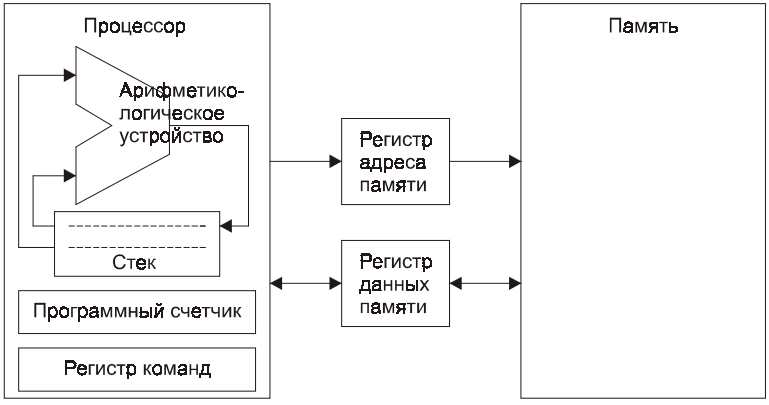


Рис. 6.7. Стековая архитектура

Так в данной архитектуре выглядит последовательность машинных команд, которая может использоваться для сложения двух чисел:

```
push   1000 /* положить на вершину стека значение по адресу 1000 */
push   1002 /* положить на вершину стека значение по адресу 1002 */
add         /* снять два операнда с вершины стека, сложить их, и
               результат положить на вершину */
pop    1003 /* результат, находящийся на вершине стека, поместить
               по адресу 1003 */
```

6.2.2. Архитектуры CISC, RISC и VLIW

Врач может похоронить свою ошибку, архитектор — разве что обсадить стены плющом.

Франк Ллойд Райт

Предваряя обзор архитектур CISC, RISC и VLIW еще раз подчеркнем, что они обозначают идеализированные концепции. Большинство широко известных реальных процессоров занимают промежуточное положение на шкале, левый край которой может быть помечен концепцией CISC, а правый — RISC. Если вы захотите поэкспериментировать с размещением процессоров на такой шкале, то рекомендуем "выйти на плоскость", используя еще одну шкалу на перпендикулярной оси — разрядность процессора.

6.2.2.1. Компьютер с полным набором команд

Основоположником архитектуры, называемой *компьютер с полным набором команд* (Complete Instruction Set Computer — CISC), считают компанию IBM с ее базовой архитектурой IBM/360, ядро которой используется с 1964 года. К классическим CISC-архитектурам можно отнести архитектуру VAX. Микропроцессоры компании Intel (архитектурный ряд x86 и Pentium) достаточно близки к данной архитектуре.

Для CISC-процессоров характерно следующее:

- сравнительно небольшое число регистров общего назначения (16 регистров у классических CISC-архитектур);
- большое количество машинных команд, некоторые из которых нагружены семантически аналогично операторам высокоуровневых языков программирования и выполняются за несколько тактов;
- большое количество методов адресации;
- большое количество поддерживаемых форматов команд различной разрядности;
- преобладание двухадресного формата команд.

6.2.2.2. Компьютер с сокращенным набором команд

Архитектура, называемая *компьютер с сокращенным набором команд* (Reduced Instruction Set Computer — RISC), появилась благодаря тому, что еще в середине 70-х годов XX века некоторые разработчики компьютерных архитектур заметили, что даже у компьютеров сложной архитектуры бóльшая часть времени уходит на выполнение простых команд. Справедливым оказалось правило 20/80, а именно: 20 % команд используется в 80 % случаев, а оставшиеся 80 % команд — в 20 % случаев. Это наблюдение легло в основу работ по созданию IBM 801 — первой RISC-машины (рис. 6.8), разработка которой была завершена к 1979 году.

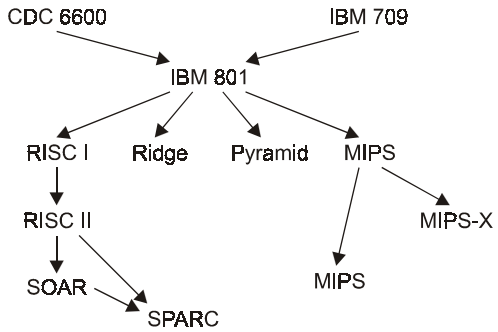


Рис. 6.8. История развития RISC-архитектуры

Само понятие RISC было введено Дэвидом Паттерсоном (David Patterson), преподавателем университета Беркли, в 1980 году.

Основными чертами концепции RISC-архитектуры являются:

- одинаковая длина команд;
- единый формат команд. Или, по крайней мере, использование не более двух-трех форматов;
- операндами всех арифметических и логических команд могут быть только регистры;
- выполнение командой только простых действий;
- выполнение любой команды производится не дольше, чем за один такт;
- большой регистровый файл;
- только простая адресация.

В системах программирования для RISC-архитектуры практически всегда присутствуют "умные" компиляторы, имеющие большие возможности по оптимизации кода.

6.2.2.3. Архитектура с длинным командным словом

Архитектура *с длинным командным словом* (Very Long Instruction Word — VLIW) — это статическая суперскалярная архитектура. Несколько простых команд упаковывается компилятором в длинное слово. Слово соответствует набору функциональных устройств. Распараллеливание кода производится на этапе компиляции, и в машинном коде уже присутствует явный параллелизм.

Примером реализации VLIW-архитектуры является суперкомпьютер Эльбрус-3. Собственно длинное командное слово занимает 256 бит в упакованном виде и до 500 бит в распакованном представлении. Микрораспараллеливание на уровне операций обеспечивается как для скалярных, так и для векторных вычислений. В каждой команде в максимальном варианте может запускаться одновременно до 7 арифметико-логических операций. В качестве аргументов они могут использовать результаты ранее выполненных операций, поступающие непосредственно с выхода арифметико-логических устройств либо из быстрой регистровой памяти (буферы стека емкостью 1024 слова или буферы считанных элементов массивов емкостью 512 слов). Предусмотрено одновременное параллельное обращение по 8 каналам в локальную и (или) глобальную память.

Развитием данной архитектуры являются вычисления с явным параллелизмом команд (Explicitly Parallel Instruction Computing — EPIC). Концепция EPIC разработана совместно компаниями Intel и Hewlett-Packard. Она обладает достоинствами VLIW, но лишена ее недостатков (например, использует специальные механизмы для исключения неэффективности кодов традиционных VLIW-архитектур, требовавших применения пустых команд для заполнения пустых машинных тактов). Данная архитектура имеет следующие основные особенности:

- масштабируемость архитектуры до большого количества функциональных устройств;
- явный параллелизм в машинном коде;
- предикатное выполнение команд, исключающее переходы. Команды из разных частей условного ветвления снабжаются предикатными полями и запускаются параллельно.

6.2.3. Вычисления

> Целые числа сотворил господь бог, все остальное — дело рук человеческих.
>
> *Леопольд Кронекер*

Выполнение математических расчетов на вычислительных системах связано с использованием вещественной машинной арифметики (т. е. арифметики с плавающей точкой). Поскольку любое вещественное число в машинном представлении определено дискретно, то точность таких вычислений сильно

зависит от машинного представления данных. Для того чтобы полученные результаты имели практическую ценность, следует иметь некоторые оценки их точности.

6.2.3.1. Формат чисел с плавающей точкой IEEE 754

В мире есть 10 категорий людей: те, кто знает двоичное исчисление, и те, кто не знает.

Программистский фольклор

На многих архитектурных платформах в качестве стандарта принят формат чисел с плавающей точкой IEEE 754. Принципиальными различиями между типами чисел с плавающей точкой являются точность представления, диапазон представления и требуемая для их хранения память. В табл. 6.1 показаны эти отношения.

Таблица 6.1. *Типы чисел с плавающей точкой*

Тип числа	Количество значащих цифр	Количество байт
Обычная точность	6—7	4
Двойная точность	15—16	8

Числа с плавающей точкой представляются в виде мантиссы, содержащей значение числа, и экспоненты, содержащей порядок степени числа. В табл. 6.2 приведено количество бит, необходимое для представления мантиссы и экспоненты каждого из типов. Старший бит любого числа с обычной или двойной точностью всегда является битом знака. Если старший бит установлен в единицу, число является отрицательным, если в ноль — положительным.

Таблица 6.2. *Длины экспонент и мантисс*

Тип числа	Длина экспоненты	Длина мантиссы
Обычная точность	8 бит	23 бита
Двойная точность	11 бит	52 бита

Многие алгоритмы компьютерной алгебры предполагают, что все коэффициенты формул и уравнений представляют собой точные вещественные числа. Однако в машинном представлении вещественные числа могут быть представлены лишь приближенно. Результирующая ошибка округления отражается на точности представления коэффициентов окончательного результата. Дополнительным недостатком является то, что нет

непосредственной возможности оценить погрешность представления окончательного результата.

6.2.3.2. Интервальная арифметика

При практических вычислениях приходится сталкиваться со следующими проблемами.

- *Проблема погрешности начальных данных.* В реальной жизни входные данные никогда не являются абсолютно точными.
- *Проблема представления данных.* Вещественные числа всегда представляются в машине округленно.

Интервальная арифметика предлагает единый подход к решению этих проблем. Используя его можно добиться вычисления результата с известной погрешностью, достаточной для данной задачи. Теоретически погрешность результата при применении какого-либо алгоритма зависит от погрешности начальных данных. Практически погрешность увеличивается за счет неточности представления данных.

Интервальная арифметика, в отличие от обычной арифметики, определена на множестве интервалов (**http://www.mscs.mu.edu/~globsol/walster-papers.html**). Обозначим **x**=[x, x'] и **y**=[y, y']. Тогда четыре элементарных операции идеализированной интервальной арифметики могут быть описаны следующим образом:

x + **y** = [x + y, x' + y']

x − **y** = [x − y', x' − y]

x * **y** = [min{x'y', xy', x'y, xy}, max{ x'y', xy', x'y, xy}]

1 / **x** = [1/x', 1/x] (для x не равного нулю)

Операция деления представляется как операция умножения одной переменной на величину обратную другой:

x / **y** = **x** * (1/**y**)

В результате область значений каждой из четырех элементарных операций интервальной арифметики есть точный диапазон значений соответствующей операции вещественной арифметики. В случае композиции элементарных операций могут быть получены границы диапазона значений вещественной функции.

Например, если f(x) = x(x − 1), то:

f([0, 1]) = [0, 1]([0, 1] − 1) = [0, 1][-1, 0] = [-1, 0].

Результат содержит точный диапазон [-1/4, 0].

Мощность интервальной арифметики определяется ее поддержкой вычислительной системой. Как правило, применяется следующий подход:

- использование внешнего округления дает строгие границы для диапазонов операций и функций;

- результаты вычислений представляют собой интервалы, в которых должно лежать точное значение, что очень важно для научных расчетов.

> ***О погрешностях***
>
> Интервальная арифметика дает оценки максимальной погрешности. Они мало реальны, особенно для деления при границах интервала разных знаков. Гораздо реальнее математические ожидания погрешности данных, растущие значительно медленнее. Грубая оценка роста погрешности `O(sqrt(n))`, а не `O(n)`. Существуют подходы к вычислению таких оценок по ходу исполнения программы.

6.2.4. Архитектура ЭВМ и языки программирования

Гленфордом Майерсом [Майерс 1985] исследован семантический разрыв между принципами, лежащими в основе языков высокого уровня, и принципами, определяющими архитектуру вычислительных систем. Объекты манипулирования и соответствующие им операции архитектуры вычислительной системы редко имеют близкое родство с объектами и операциями языков программирования. Для подробного понимания разрыва следует взять некоторый язык и какую-либо архитектуру и рассмотреть взаимосвязи между ними. Далее следует определить принципы, положенные в основу языка программирования, и попытаться оценить "расстояние" между ними и соответствующими принципами, на которых базируется архитектура вычислительной системы.

Компиляторы и их возможности по оптимизации и распараллеливанию программ играют решающую роль в современных архитектурах.

6.2.5. Оценка производительности вычислительных систем

Компьютер, который выполнит один и тот же объем работы за меньшее количество времени, является более быстрым. Основная единица измерения производительности компьютера — время.

С программой могут быть связаны следующие типы времени.

- *Время выполнения* (астрономическое, ответа, прошедшее), являющееся полным временем выполнения программы.
- *Время центрального процессора*, не включающее время выполнения других программ параллельно на том же процессоре или время ввода-вывода. Это время является суммой двух времен:
  - *пользовательское время*, потраченное процессором непосредственно на выполнение программы пользователя;
  - *системное время*, затраченное операционной системой на выполнение заданий, затребованных программой.

Обычно при измерениях производительности используется именно время центрального процессора, поскольку на разных системах некоторые задания (например, математические функции) могут либо быть частью программы пользователя, либо являться частью стандартной библиотеки операционной системы.

В большинстве современных процессоров скорость протекания процессов задается единой системой сигналов синхронизации. Дискретные временные события называются *тактами синхронизации*. Время центрального процессора для некоторой программы может быть получено умножением количества тактов синхронизации на время одного такого такта. Важной характеристикой процессоров является количество тактов синхронизации на одну команду. Таким образом, производительность процессора зависит от следующих параметров:

- такта синхронизации;
- среднего количества тактов синхронизации на команду;
- количества выполняемых команд.

Существуют еще две популярные единицы измерения производительности.

- *Миллион команд в секунду* (Million Instruction Per Second — MIPS). Это отношение количества команд в программе ко времени ее выполнения. Более быстрый процессор характеризуется большим числом MIPS. Основной недостаток данной единицы измерения в том, что она зависит от набора команд процессора и затрудняет сравнение процессоров с разными системами команд.
- *Миллион элементарных арифметических операций над числами с плавающей точкой в секунду* (Million FLoating point Operation Per Second — MFLOPS). Эта единица измерения предназначена для оценки только операций с плавающей точкой и не применима вне этой области. Если программа редко использует арифметику с плавающей точкой, то ее MFLOPS имеет значение, близкое к нулю.

Разработаны специальные тестовые пакеты, на которых измеряется производительность процессоров. Рассмотрим некоторые из них.

- Ливерморские циклы — набор фрагментов программ на языке FORTRAN. Фрагменты взяты из реальных программных систем и считаются типичными для численных задач.
- LINPACK — пакет программ на языке FORTRAN для решения систем линейных алгебраических уравнений.
- SPECint92 и SPECfp92 — пакеты, базирующиеся на реальных прикладных программах из различных предметных областей. Для разработки и публикации набора таких тестов была создана специальная корпорация SPEC (Standard Performance Evaluation Corporation).

6.3. Организация вычислительной системы

6.3.1. Процессор

Процессор — это программно-управляемое устройство, предназначенное для обработки цифровой информации и управления процессом этой обработки. Процессор выполняет команды и содержит два обязательных компонента.

- *Арифметико-логическое устройство* — узел, выполняющий арифметические и логические операции над операндами. Основные арифметические операции — это сложение, вычитание, умножение и деление двух чисел. Основные логические операции — отрицание, конъюнкция, дизъюнкция и эквивалентность.
- *Устройство управления* — узел, вырабатывающий и опознающий команды, осуществляющий управляемое переключение схем и цепей передачи данных, а также включающий устройства контроля.

Процессор характеризуется:

- архитектурой набора команд. Поскольку именно архитектура набора команд является одним из составляющих элементов архитектуры компьютера, то процессор играет определяющую роль в архитектуре компьютера;
- скоростью выполнения команд *(см. разд. 6.2.5)*;
- длиной машинного слова, которое может занимать одну или несколько ячеек (байтов) памяти. Машинное слово воспринимается арифметическим устройством, устройством управления или памятью как единое целое, имеющее определенное смысловое содержание. Часто смысл слова сводится к возможности расчленения его на части (поля). Этот смысл оказывается разным, в зависимости от того, в какое устройств поступило (для какого устройства было предназначено) данное слово.

Выделяют два режима работы процессора.

- *Привилегированный режим (режим ядра операционной системы)*, при котором разрешены привилегированные команды (например, команды управления памятью).
- *Непривилегированный (пользовательский) режим*, при котором привилегированные команды не разрешены.

6.3.2. Память

Память — это функциональная часть компьютера, предназначенная для записи, хранения и считывания информации. Технически память реализуется в виде запоминающих устройств.

В самом простейшем случае можно выделить основную память и внешнюю. На рис. 6.9 представлены основные классы памяти, которые могут присутствовать на современных компьютерах.

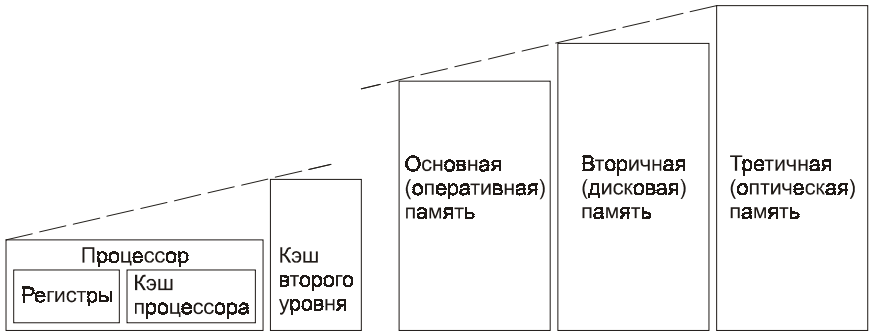


Рис. 6.9. Иерархия классов памяти

В основе иерархии памяти современных компьютеров лежат два принципа.

- *Принцип локальности обращений.* Большинство запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации, как временной, так и пространственной.
- *Соотношение скорость доступа — объем памяти.* Чем быстрее может быть осуществлен доступ к памяти, тем она более дорогая. Следовательно, тем менее она практична с точки зрения использования и тем меньше будет ее объем, который разумно устанавливать в компьютер. Некоторые числовые данные по средней скорости доступа к памяти и ее типичному объему представлены в табл. 6.3.

Таблица 6.3. *Сравнение характеристик различных классов памяти*

Класс памяти	Доступ (в наносекундах)	Объем (в килобайтах)
Кэш процессора	3	32
Кэш 2-го уровня	30	4096
Основная память	300	500 000
Дисковая (вторичная) память	30 000 000	10 000 000

6.3.3. Шина

Шина служит для обмена информацией между отдельными функциональными устройствами компьютера. Сигналы, подаваемые на шину, разделяются на четыре категории [Петцольд 2001].

- *Адресные* — генерируемые процессором и обычно использующиеся для адресации оперативной памяти. Могут также применяться для обращения к другим устройствам компьютера.

- *Вывода данных* — также генерируемые процессором. Используются для передачи данных в память и на другие устройства.
- *Ввода данных* — генерируемые различными устройствами компьютера (например, памятью) и поступающие в процессор.
- *Управляющие* — генерируемые процессором и другими устройствами, которым нужно что-то сообщить процессору. Примером является сигнал, с помощью которого процессор указывает на необходимость записи данных в определенную ячейку памяти.

6.3.4. Устройства ввода и вывода данных

Программист читает внуку сказку:
— "...стал он кликать золотую рыбку..."
— Деда, а почему рыбку?
— А потому, дружок, что мышек тогда еще не было.
Программистский фольклор

Устройства ввода-вывода предназначены для обмена информацией между компьютером и окружающим миром. Этот термин часто употребляется в собирательном смысле для обозначения любого из устройств ввода или вывода. Данные устройства естественным образом подразделяются на две категории.

- *Устройства ввода*, позволяющие осуществлять ввод данных в компьютер.
- *Устройства вывода*, позволяющие осуществлять вывод результатов обработки данных, программ и другой информации, хранящейся в компьютере. Устройства вывода преобразуют данные, хранящиеся в цифровой форме, в форму, которая удобна для человеческого восприятия (например, текст, графика, звук).

Из всей совокупности устройств ввода-вывода подчеркнем особую важность монитора и клавиатуры, как устройств, ориентированных на прямое общение человека с компьютером.

6.3.5. Функциональное описание

Вычислительные системы могут иметь самые разные конфигурации. *Функциональное описание* отражает основное функциональное назначение составных частей компьютера и связи между ними.

Эффективность использования компьютеров и потенциальные возможности развития компьютерной системы в будущем существенно зависят от конфигурации. В первых компьютерах каждое функциональное устройство имело специальный набор контактных связей со всеми другими функциональными устройствами компьютера. В настоящее время достаточно часто объединение функциональных блоков осуществляется посредством системы шин.

Впервые этот способ был предложен компанией DEC. Шины — это простые коммутаторы с разделением времени. Существуют и более сложные коммутаторы, в том числе и составные *(см. разд. 6.5.2)*.

Сейчас естественным является такое положение, что разные экземпляры компьютеров одного и того же семейства могут быть выполнены на разной элементной базе и могут иметь различное конструктивное оформление. Независимо от этого все компьютеры данной архитектуры (с той же самой системой команд) полностью совместимы между собой. Поэтому прикладные программы, разработанные для одной модели компьютера, можно переносить на компьютеры другой модели, но обязательно принадлежащей тому же самому семейству.

Заметим, что общих подходов для решения задачи выбора конфигурации системы не существует. Тем не менее, существует *принцип достаточности конфигурации* в целом и параметров ее частей для решения типовых задач пользователя. Например, в случае использования компьютера в режиме пишущей машинки, быстродействие процессора может мало волновать пользователя. Если же пользователь переключается на компиляцию программ, написанных, например, на языке C++, снижение быстродействия в десятки или сотни раз будет весьма досадным.

6.4. Особенности SPARC-архитектуры

Число различных реальных архитектур исчисляется сотнями, а процессоров, реализующих архитектуры, — тысячами. В сети Интернет можно найти два подробных обзора современных микропроцессоров:

- обзор Олега Репина (**http://www.microprocessor.sscc.ru**);
- обзор Марка Смозермана (Mark Smotherman) (**http://www.cs.clemson.edu/~mark/architects.html**).

В этом разделе мы дадим краткую информацию об истории и особенностях одной из наиболее интересных и элегантных архитектур — масштабируемой процессорной архитектуре SPARC (Scalable Processor ARChitecture). Также мы приведем основные сведения о командах языка Assembler для данной архитектуры и рассмотрим пример программы, порожденный компилятором.

6.4.1. История создания SPARC-архитектуры

SPARC-архитектура была сформулирована в 1985 году компанией Sun Microsystems, Inc. Основные идеи были взяты из архитектур RISC I, RISC II и SOAR, разработанных инженерами университета Беркли в начале 80-х годов XX века (см. рис. 6.9).

С одной стороны, SPARC-архитектура обладает всеми особенностями RISC-архитектур. С другой стороны, у нее есть свои специфические особенности:

- три основных формата команд;
- небольшое число моделей адресации;
- достаточно большой регистровый файл;
- отдельный регистровый файл для чисел с плавающей точкой;
- определение архитектурой наличия сопроцессора.

При разработке данной архитектуры решались следующие основные задачи:

- поддержка языка программирования C и операционной системы Unix;
- поддержка вычислительных приложений (большей частью написанных на языке FORTRAN);
- поддержка приложений искусственного интеллекта и экспертных систем (написанных на языках Lisp и Prolog).

Первоначально SPARC-архитектура разрабатывалась с целью упрощения реализации 32-битного процессора. Наиболее известной 32-разрядной версией этой архитектуры является SPARC-V8, в которой под виртуальные адреса и целые числа отводится 32 бита. По мере развития технологии интегральных схем эта архитектура совершенствовалась и стала 64-разрядной в версии SPARC-V9.

6.4.2. Основные понятия SPARC-архитектуры

В этом разделе мы рассмотрим такие базисные понятия SPARC-архитектуры, как регистры, регистровые окна и стек, а также общий синтаксис команд и модель адресации.

6.4.2.1. Регистры и регистровые окна

Начнем изучение языка Assembler с регистров. В табл. 6.4 приведены стандартные обозначения для регистров общего назначения. Именно такие обозначения следует использовать в программах.

Таблица 6.4. *Обозначения для регистров общего назначения в SPARC-архитектуре*

Регистр	Назначение
`%r`*i*	Регистр для целых чисел с номером *i* (0 ≤ *i* ≤ 31)
`%g`*i* (`%r0-%r7`)	Глобальный регистр для целых чисел с номером *i* (0 ≤ *i* ≤ 7)

Таблица 6.4 *(окончание)*

Регистр	Назначение
%i*i* (%r24-%r31)	Входной регистр для целых чисел с номером *i* (0 ≤ *i* ≤ 7)
%l*i* (%r16-%r23)	Локальный регистр для целых чисел с номером *i* (0 ≤ *i* ≤ 7)
%o*i* (%r8-%r15)	Выходной регистр для целых чисел с номером *i* (0 ≤ *i* ≤ 7)
%f*i*	Регистр для чисел с плавающей точкой с номером *i* (0 ≤ *i* ≤ 31)
%sp	Регистр — указатель на вершину стека (%o6)
%fp	Регистр — указатель на фрейм (предыдущая вершина стека) (%i6)

Существуют несколько особенностей, связанных с регистрами, которые мы перечислим в виде списка.

- Глобальный регистр %g0 всегда содержит константу 0. Содержимое этого регистра не может быть изменено, поскольку запись в него игнорируется.
- По соглашению в регистры %i7 и %o7 помещаются адреса возврата. Программист не должен использовать их для других целей.
- По соглашению в регистры %i6 и %o6 помещаются указатели на вершины стека и фрейма. Программист не должен использовать их для других целей.
- Существует единственный набор глобальных регистров, который доступен всем подпрограммам программы.

В любой момент времени команде языка Assembler доступны 8 глобальных (global) целочисленных регистров и регистровое окно из 24-х регистров. Регистровое окно образуют 8 входных (in) %i*i*, 8 локальных (local) %l*i* и 8 выходных (out) %o*i* регистров. Восемь выходных регистров одновременно являются входными регистрами следующего окна. Регистровые окна используются для эффективной передачи параметров в вызываемую процедуру из вызывающей и наоборот (рис. 6.10). Количество окон зависит от реализации и находится в диапазоне от 3-х до 32-х. Кроме целочисленных регистров, каждой процедуре доступны глобальные регистры с *плавающей точкой* (floating point).

В SPARC-архитектуре существуют более 20 регистров специального назначения. Некоторые из них приведены в табл. 6.5.

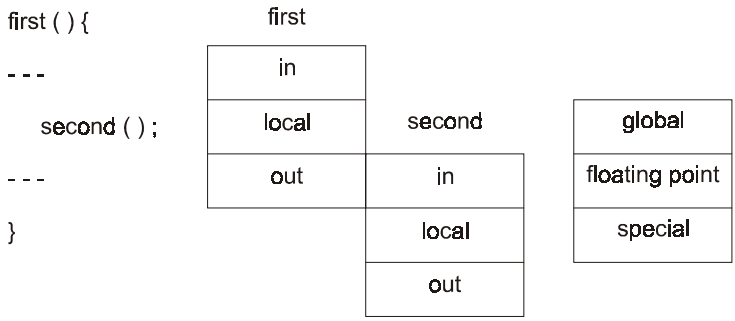


Рис. 6.10. Регистровые окна в SPARC-архитектуре

Таблица 6.5. *Регистры специального назначения в SPARC-архитектуре*

Регистр	Назначение
`%PC` (Program Counter)	Содержит адрес исполняемой в данный момент команды
`%NPC` (Next Program Counter)	Содержит адрес следующей команды
`%PSR` (Processor State Register)	Содержит большое количество полей с кодами условий

6.4.2.2. Общий синтаксис команд и модель адресации

Команда языка Assembler состоит из:

- необязательной метки, заканчивающейся двоеточием;
- кода операции;
- последовательности операндов, разделенных запятыми. Операнд, в который помещается результат, всегда указывается последним;
- необязательного комментария, начинающегося с восклицательного знака.

Существуют два вида комментариев: первый начинается с восклицательного знака и завершается концом строки, а второй помещается между символами '`/*`' и '`*/`'.

Большинство команд может быть отнесено к одной из трех следующих категорий (табл. 6.6).

Таблица 6.6. *Вид команд в SPARC-архитектуре*

Категория команд	Вид команды
Работы с памятью	`opcode [%reg1+const],%reg2` (или `opcode [%reg1+%reg2],%reg3`)
Арифметические, логические, сдвиги	`opcode %reg1,const,%reg2` (или `opcode %reg1,%reg2,%reg3`)
Переходы	`opcode address`

Фактически мы обращаемся к памяти, используя либо результат сложения двух регистров, либо регистра и 13-битной знаковой константы. В листинге 6.2 приведено несколько примеров использования моделей адресации.

Листинг 6.2. Примеры использования моделей адресации

```
Ld      [%fp+68],%o0     ! Загрузить в регистр %o0
                         ! значение по адресу [%fp+68]
st      %o0,[%fp-4]      ! Содержимое регистра %o0
                         ! запомнить по адресу [%fp-4]
add     %i4,%o0,%i5      ! Сложить содержимое регистров %i4 и %o0,
                         ! результат поместить в %i5
add     %i4,3,%i4        ! Добавить 3 к содержимому регистра %i4
```

6.4.2.3. Стек

Стек (рис. 6.11) предназначен для хранения значений автоматических и временных переменных и возвращаемой информации для каждого вызова процедуры. Когда происходит вызов процедуры, на стеке отводится область, называемая *фреймом* (frame). Эта область освобождается при выходе из процедуры. Вершина стека (она обозначается как `%sp`) растет в сторону уменьшения адресов.

6.4.3. Основной набор команд языка Assembler

В этом разделе мы приведем основной набор команд языка Assembler [SPARC 1994].

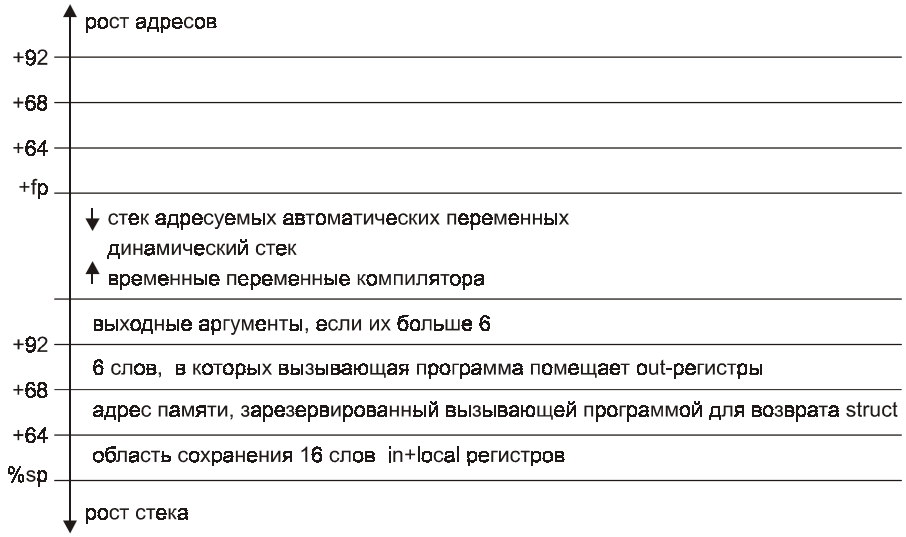


Рис. 6.11. Стек в SPARC-архитектуре

6.4.3.1. Команды работы с памятью

Один программист другому: "Вот представь, что у тебя есть 1000 рублей...
Или, для круглого счета, пусть у тебя 1024..."

Программистский фольклор

В RISC-архитектурах обычно имеется очень небольшое количество команд работы с памятью. В SPARC-архитектуре существуют всего два типа таких команд.

- Команды пересылки из памяти в регистр (команды загрузки).
- Команды пересылки из регистров в память (команды запоминания).

Список таких команд приведен в табл. 6.7.

Таблица 6.7. *Команды работы с памятью*

Команда	Выполняемое действие
`ldub`	Загрузить беззнаковый байт
`ldsb`	Загрузить знаковый байт
`lduh`	Загрузить беззнаковое полуслово
`ldsh`	Загрузить знаковое полуслово
`ld`	Загрузить слово (32-битное)
`ldd`	Загрузить двойное слово

Таблица 6.7 *(окончание)*

Команда	Выполняемое действие
`stb`	Запомнить байт по указанному адресу
`sth`	Запомнить полуслово по указанному адресу
`st`	Запомнить слово по указанному адресу
`std`	Запомнить двойное слово по указанному адресу

6.4.3.2. Арифметические и логические команды

Основные арифметические и логические команды (табл. 6.8) близки к аналогичным командам большинства других архитектур. Для того чтобы команды устанавливали бит кода условия, следует указывать суффикс `cc` после кода команды (например, `subcc`).

Таблица 6.8. *Арифметические и логические команды*

Команда	Выполняемое действие
`add`	Сложение
`sub`	Вычитание
`and`	Побитовое "И"
`andn`	Побитовое "И" с инвертированным вторым операндом
`or`	Побитовое "ИЛИ"
`orn`	Побитовое "ИЛИ" с инвертированным вторым операндом
`xor`	Побитовое исключающее "ИЛИ"
`xnor`	Побитовое исключающее "ИЛИ" с инвертированным вторым операндом

6.4.3.3. Команды *sethi* и *nop*

Команда `sethi` помещает 22-битную константу `const` в 22 старших бита регистра `%reg`:

```
sethi  const,%reg
```

При этом 10 младших битов регистра `%reg` устанавливаются в ноль. Существуют два специальных оператора языка Assembler: `%hi()` и `%lo()`, которые извлекают соответственно 22 старших бита и 10 младших битов операнда,

указанного в скобках. Для того чтобы поместить 32-битную константу `L` (например, некоторый определенный адрес) в регистр `%o0`, следует выполнить следующую последовательность команд:

```
sethi  %hi(L),%o0
or     %o0,%lo(L),%o0
```

Поскольку такой способ загрузки константы применяется довольно часто, язык Assembler определяет команду `set`. Команда с параметрами `set L,%o0` является комбинацией двух предыдущих команд.

Обратим внимание и на то, что загрузка младших битов обычно комбинируется с командой `st`, что позволяет сразу помещать по указанному адресу содержимое некоторого регистра (например, `%l0`):

```
sethi  %hi(L),%o0
st     %l0,[%o0+%lo(L)]
```

Ничего не делающая команда `nop` (no operation) на самом деле представляет собой `sethi  0,%g0`.

6.4.3.4. Команды *save* и *restore*

Команды `save` и `restore` выполняют две основные задачи.

- Переключение на следующее (и, соответственно, предыдущее) регистровое окно.
- Отведение (и, соответственно, освобождение) памяти на стеке.

Фактически команды `save` и `restore` эквивалентны команде `add`, дополненной возможностью переключения окон. Типичный пример использования команды `save`, при котором произойдет переключение на следующее регистровое окно и на стеке будет отведено 96 байтов, выглядит так:

```
save   %sp,-96,%sp
```

6.4.3.5. Команды переходов

Основные команды переходов приведены в табл. 6.9.

Таблица 6.9. *Команды переходов*

Команда	Выполняемое действие
`ba`	Безусловный переход
`be`	Переход по "равно"
`bne`	Переход по "не равно"

Таблица 6.9 *(окончание)*

Команда	Выполняемое действие
`bg`	Переход по "больше"
`bge`	Переход по "больше или равно"
`bl`	Переход по "меньше"
`ble`	Переход по "меньше или равно"
`call`	Вызов подпрограммы
`jmpl`	Косвенный вызов с адресом в регистре

Все переходы исполняются только после того, как будет исполнена идущая за переходом команда (называемая также отложенной командой). Это достаточно интересная и эффективная особенность архитектуры, поскольку переходы не изменяют регистр `%PC`, а изменяют регистр `%NPC`, в который уже считана отложенная команда. В качестве отложенной команды рекомендуется использовать содержательную команду, учитывая, что она будет исполнена до команды перехода. В крайнем случае, рекомендуется применять команду `nop`. При записи команд иногда используют следующее полезное соглашение: отложенную команду сдвигают на один пробел вправо.

Для вызова подпрограмм используется команда `call` с единственным параметром — адресом вызываемой подпрограммы. Команда передает управление по указанному адресу. Существует соглашение, согласно которому параметры вызываемой подпрограммы должны находиться в регистрах от `%o0` до `%o5`, а возвращаемый подпрограммой скалярный результат будет помещен в регистр `%o0`.

6.4.3.6. Команды синхронизации

Команды синхронизации представляют собой атомарные команды, что означает их неделимость, невозможность прерывания в процессе их исполнения. На многопроцессорных компьютерах гарантируется, что если на два или более процессоров эти команды поступят одновременно, то они будут исполнены в неопределенном, но последовательном порядке. Приведем примеры двух таких команд.

- Команда `ldstub` (load-store unsigned byte) копирует содержимое памяти (байт) по адресу `addr` в регистр `reg`, а все биты памяти `addr` устанавливаются в единицу:

```
ldstub [addr],reg.
```

- Команда `swap` обменивает содержимое регистра `reg` и содержимое памяти (слово) по адресу `addr`:

```
swap   [addr],reg.
```

6.4.3.7. Синтетические команды

В SPARC-архитектуре (как и в любой другой RISC-архитектуре) отсутствуют многие команды, которые можно встретить на других архитектурах. С целью обеспечения удобства SPARC-Assembler имеет большое количество синтетических команд, которые он транслирует в эквивалентные команды, типичные для SPARC-архитектуры. Некоторые синтетические команды приведены в табл. 6.10.

Таблица 6.10. *Синтетические команды*

Синтетическая команда	Эквивалентная команда
`cmp %reg,%reg`	`subcc %reg,%reg,%g0`
`dec %reg`	`sub %reg,1,%reg`
`inc %reg`	`add %reg,1,%reg`
`mov %reg,%reg`	`or %g0,%reg,%reg`
`neg %reg`	`sub %g0,%reg,%reg`
`not %reg`	`xnor %reg,%g0,%reg`
`restore`	`restore %g0,%g0,%g0`
`ret`	`jmpl %i7+8,%g0`

6.4.3.8. Псевдокоманды языка Assembler

Псевдокоманды (или директивы) языка Assembler выполняются во время трансляции программы и, как правило, не порождают команд. Все псевдокоманды начинаются с точки. Некоторые из них приведены в табл. 6.11.

Таблица 6.11. *Псевдокоманды языка Assembler*

Псевдокоманда	Назначение
`.section`	Начало сегмента
`.size`	Установить размер символа (в байтах)
`.type`	Установить тип символа
`.align`	Установить выравнивание (в байтах)
`.skip`	Захватить указанное количество байтов в текущем сегменте
`.global`	Глобальный символ, доступен внешним программам
`.word`	Константа длиной слово
`.double`	Константа длиной двойное слово

6.4.4. Примеры программ на языке Assembler

Начнем этот раздел с достаточно очевидного утверждения: писать программы на языке Assembler, в том числе и на SPARC-Assembler, неразумно. Современные оптимизирующие компиляторы генерируют код, сравнимый по эффективности с кодом, написанным вручную. Тем не менее, в некоторых случаях знание языка Assembler является полезным, а в ряде случаев (например, при разработке компиляторов) просто необходимым.

6.4.4.1. Простейшая программа

Перед тем как написать простейшую программу, укажем некоторые минимальные требования и основные соглашения о связях.

- Следует отвести не менее 92-х байтов (23 32-разрядных слова) на стеке. Эту память может использовать для сохранения значений механизм регистровых окон (например, при прерывании по нехватке очередного регистрового окна). Реально должно быть отведено 96 байтов, поскольку память должна быть выровнена на границу двойного слова.
- Точкой входа в программу пользователя является глобальный символ `main`. Так выглядит возможное начало программы:

```
.global main
main:      save   %sp,-96,%sp.
```

- Для возврата управления операционной системе следует применять две следующие команды:

```
ret      ! Стандартная команда возврата из подпрограммы
 restore ! Вернуться к предыдущему регистровому окну
         ! Заметим, что restore – отложенная команда
         ! и будет выполнена до ret.
```

Текст простейшей программы, печатающей сообщение "`Hello, world!`", приведен в листинге 6.3.

Листинг 6.3. Текст программы, печатающей сообщение "`Hello, world!`"

```
! Минимальный пролог секции кода
       .section      ".text"
       .global       main
       .align 4
main:
       save   %sp,-96,%sp
```

```
! Основная часть: вызов подпрограммы (printf) для печати строки
       set    string,%o0 ! Комбинация команд sethi и or
       call   printf
        nop               ! Отложенная команда
! Стандартный эпилог секции кода
       ret
        restore           ! Отложенная команда
! Секция данных
       .section      ".rodata1"
       .align 4
string:
       .ascii "Hello, world!\n\000"
```

6.4.4.2. Программа, сгенерированная компилятором

Как уже отмечалось, не рекомендуется писать программы вручную на языке Assembler. По исходной программе на языке C (листинг 6.4) оптимизирующим компилятором компании Sun Microsystems, Inc. была сгенерирована программа на языке Assembler (листинг 6.5).

Листинг 6.4. Текст программы на языке C

```
#include <stdio.h>
int counter(int i) {
    return(i);
}
int main() {
    int i;
    int s = 0;
    for (i=0; i<500; i++)
        s = s + counter(i);
    printf("%d \n", s);
    return 0;
}
```

Листинг 6.5. Программа, сгенерированная компилятором

```
       .section       ".bss",#alloc,#write
```

```
Bbss.bss:
       .section       ".data",#alloc,#write
Ddata.data:
       .section       ".rodata",#alloc
!
! CONSTANT POOL
!
Drodata.rodata:
       .file   "for.c"
       .section       ".rodata1",#alloc
       .align  4
!
! CONSTANT POOL
!
.L104:
       .ascii  "%d \n\000"
       .section       ".text",#alloc,#execinstr
/* 000000        0 */        .align 4
/* 000000          */        .skip  16
/* 0x0010          */        .align 4
! FILE for.c
!    1              !#include <stdio.h>
!    2              !int counter(int i) {
!
! SUBROUTINE counter
! OFFSET    SOURCE LINE      LABEL  INSTRUCTION
                              .global counter
                           counter:
/* 000000        2 */       retl    ! Result =  %o0
/* 0x0004          */       nop
/* 0x0008        0 */       .type  counter,2
/* 0x0008        0 */       .size  counter,(.-counter)
       .section      ".text",#alloc,#execinstr
/* 000000        0 */       .align 4
!    3              !    return(i);
!    4              !}
!    5              !int main() {
!
```

```
! SUBROUTINE main
! OFFSET    SOURCE LINE      LABEL  INSTRUCTION
                              .global main
                         main:
/* 000000        5 */       save   %sp,-96,%sp
!    6              !    int i;
!    7              !    int s = 0;
/* 0x0004        7 */        or     %g0,0,%l0
!   8               !    for (i=0; i<500; i++)
/* 0x0008        8 */       or      %g0,0,%l1
!   9               !        s = s + counter(i);
/* 0x000c        9 */       or      %g0,0,%o0
                         .L900000207:
/* 0x0010        9 */       call    counter ! params =  %o0
/* 0x0014          */       add     %l1,1,%l1
/* 0x0018          */       add     %l0,%o0,%l0
/* 0x001c          */       cmp     %l1,500
/* 0x0020          */       bl      .L900000207
/* 0x0024          */       or      %g0,%l1,%o0
!   10              !    printf("%d \n", s);
                         .L77000029:
/* 0x0028       10 */       sethi   %hi(.L104),%g1
/* 0x002c          */       add     %g1,%lo(.L104),%o0
/* 0x0030          */       call    printf ! params =  %o0 %o1
/* 0x0034          */       or      %g0,%l0,%o1
!   11            !    return 0;
/* 0x0038       11 */       ret     ! Result =  %i0
/* 0x003c          */       restore %g0,0,%o0
/* 0x0040        0 */       .type   main,2
/* 0x0040        0 */       .size   main,(.-main)
```

6.4.5. Организация SPARC-архитектуры

Организацию SPARC-архитектуры мы проиллюстрируем на примере окружения архитектуры процессора UltraSPARC компании Sun Microsystems, Inc. Высокая производительность процессора UltraSPARC потребовала создания гибкой масштабируемой архитектуры межсоединений. В результате оказалось достаточно просто строить системы для широкого круга приложений.

Архитектура Ultra Port Architecture (UPA) представляет собой спецификацию, описывающую логические и физические интерфейсы порта системной шины и требования, накладываемые на организацию соединений (рис. 6.12). К этим портам подключаются все устройства системы. Архитектура UPA основана на пакетной коммутации сообщений по принципу "точка-точка".

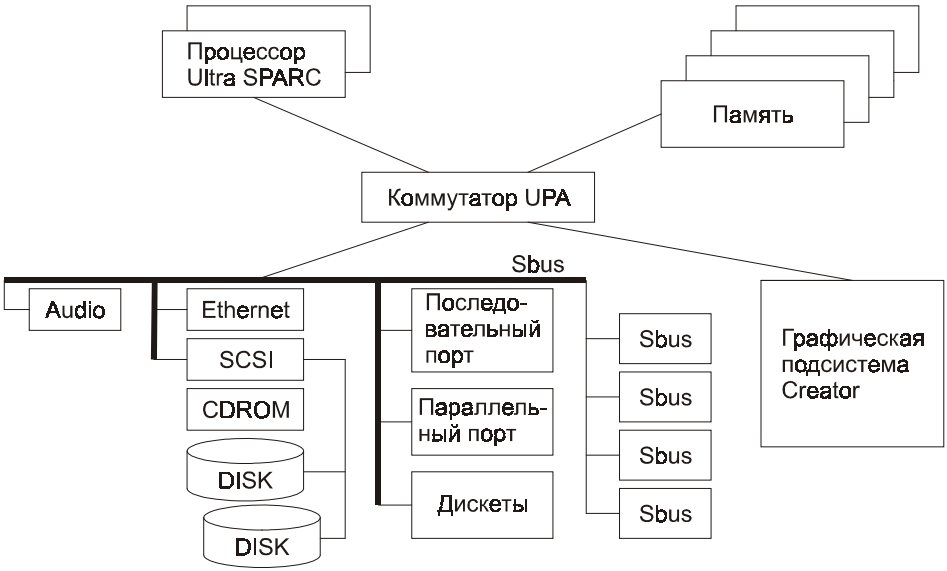


Рис. 6.12. Масштабируемость SPARC-архитектуры

Процессор UltraSPARC представляет собой высокопроизводительный, высокоинтегрированный суперскалярный процессор, реализующий архитектуру SPARC-V9 (рис. 6.13). Он имеет устройство предварительной выборки и диспетчеризации команд, целочисленное исполнительное устройство, устройство плавающей точки с графическим устройством, устройство управления памятью, устройство загрузки/записи, устройство управления внешней кэш-памятью, устройство управления интерфейсом памяти, а также внутренний кэш команд и кэш данных. Устройство предварительной выборки и дешифрации команд процессора обеспечивает выборку команд в буфер команд, окончательную их дешифрацию, группировку и распределение для параллельного выполнения в конвейерных функциональных устройствах процессора. Буфер команд имеет емкость 12 команд. Команды могут быть выбраны из любого уровня иерархии памяти: кэш-памяти команд, внешней кэш-памяти или из основной памяти системы.

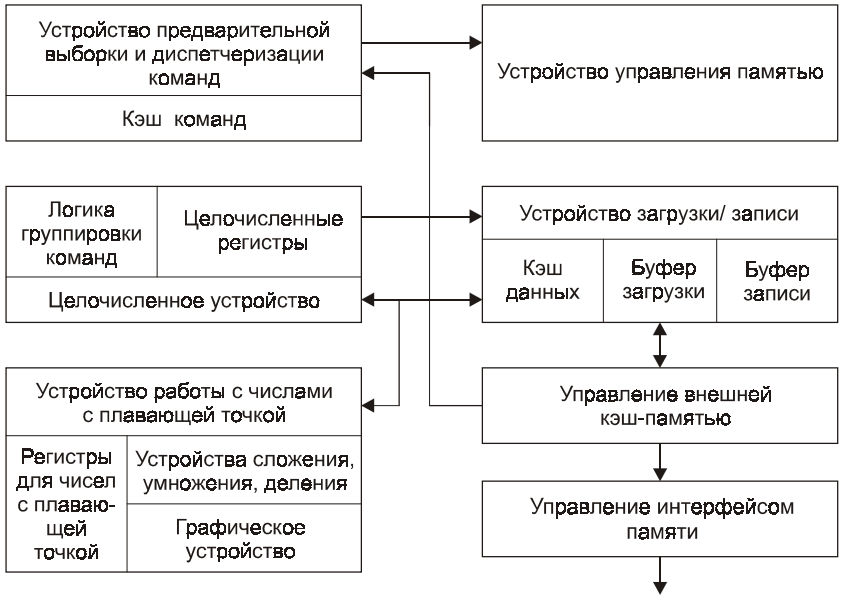


Рис. 6.13. Схема процессора UltraSPARC

6.5. Параллельные и распределенные архитектуры

6.5.1. Основные классы параллельных архитектур

Считается, что история параллельных вычислений состоит из трех основных этапов.

- Векторно-конвейерные вычисления.
- Параллельные вычисления на множестве процессоров.
- Вычисления на системах из микропроцессоров с кэш-памятью и разделяемой, физически распределенной основной памятью.

Основной характеристикой классификации параллельных и распределенных архитектур считают наличие общей или распределенной (локальной для каждого из узлов) памяти. Принято выделять пять основных классов параллельных компьютеров (**http://www.parallel.ru**).

- *Массивно-параллельные системы* (Massive Parallel Processing — MPP), состоящие из однородных вычислительных узлов, которые в свою очередь включают один или несколько центральных процессоров, локальную память, коммуникационный процессор или сетевой адаптер, жесткие

диски и другие устройства ввода-вывода. Данный класс характеризуется скоростными и специализированными каналами связи и широкими возможностями по масштабируемости (scalability).

- *Кластерные архитектуры* (Cluster Architecture), представляющие собой дешевый вариант массивно-параллельных систем. В них узлами являются обыкновенные рабочие станции или персональные компьютеры. Они связываются с помощью сетевых технологий на базе коммутатора или шинной архитектуры. Данный класс характеризуется возможностью компоновки из конфигураций симметричных мультипроцессорных систем, использованием стандартных (а следовательно, дешевых) компонентов и более слабой по сравнению с MPP масштабируемостью.
- *Симметричные мультипроцессорные системы* (Symmetric Multi-Processing — SMP), состоящие из нескольких однородных процессоров и массива общей памяти. Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью.
- *Системы с неоднородным доступом к памяти* (Non-Uniform Memory Access — NUMA) состоят из однородных базовых модулей, включающих один или несколько процессоров и блок памяти. Поддерживается единое адресное пространство, но доступ некоторых модулей к общей памяти осуществляется в несколько раз медленнее, чем к локальной. Данная архитектура представляет собой нечто среднее между архитектурами массивно-параллельных и симметричных мультипроцессорных систем.
- *Параллельные векторные системы* (Parallel Vector Processing — PVP) характеризуются наличием специальных векторно-конвейерных процессоров. В них предусмотрены команды однотипной обработки векторов независимых данных.

6.5.2. Коммутаторы вычислительных систем

Коммуникационная среда вычислительных систем на структурном уровне состоит из трех компонентов:

- *сетевых адаптеров* (сетевых плат), осуществляющих интерфейс между вычислительным устройством и сетью передачи пакетов;
- *коммутаторов сети передачи пакетов*;
- *кабелей*, служащих для подсоединения входных и выходных каналов адаптеров к портам коммутатора и соединения коммутаторов друг с другом.

Разнообразные схемы построения коммутаторов обычно комбинируют несколько их базовых типов. Коммутаторы разделяют на простые и составные, компонуемые из простых.

6.5.2.1. Простые коммутаторы

Простые коммутаторы бывают с временным и пространственным разделением [Корнеев 1999].

- Простые коммутаторы с временным разделением называются также *шинами* или *шинными структурами*. Особенность таких коммутаторов заключается в использовании общей информационной магистрали для передачи данных между устройствами, подключенными к шине (см. рис. 6.1).
- Простые коммутаторы с пространственным разделением способны соединить любой вход с любым одним выходом или подмножеством выходов. Пространственное разделение объясняется тем, что каждое соединение входов и выходов выполняется разным оборудованием. Коммутатор представляет собой совокупность мультиплексоров, число которых равно количеству выходов коммутатора.

6.5.2.2. Составные коммутаторы

Тем, кто хорошо знаком с пятым измерением, ничего не стоит
раздвинуть помещение до желательных пределов.
Скажу вам больше, уважаемая госпожа,
до черт знает каких пределов!

Михаил Булгаков. "Мастер и Маргарита"

Основной принцип построения коммутаторов с большим числом входов и выходов — создание сложного коммутатора из совокупности коммутаторов с меньшим числом входов и выходов путем объединения этих коммутаторов линиями "точка-точка", снижающих требования к размножению входов [Корнеев 1999].

Очень часто составные коммутаторы строятся из простых коммутаторов с пространственным разделением с двумя входами и двумя выходами.

Примерами составных коммутаторов являются:

- коммутатор Клоза, имеющий три каскада коммутаторов: входных, промежуточных и выходных;
- баньян-сети и дельта-сети.

Существует распределенный способ построения составных коммутаторов. При этом составной коммутатор строится из простых так, что один вход и один выход каждого составляющего коммутатора служат входом и выходом составного коммутатора. К каждому составляющему коммутатору подключается процессор и память, образуя в совокупности вычислительный модуль с каналами для соединения с себе подобными. Некоторые типы составных коммутаторов приведены на рис. 6.14.

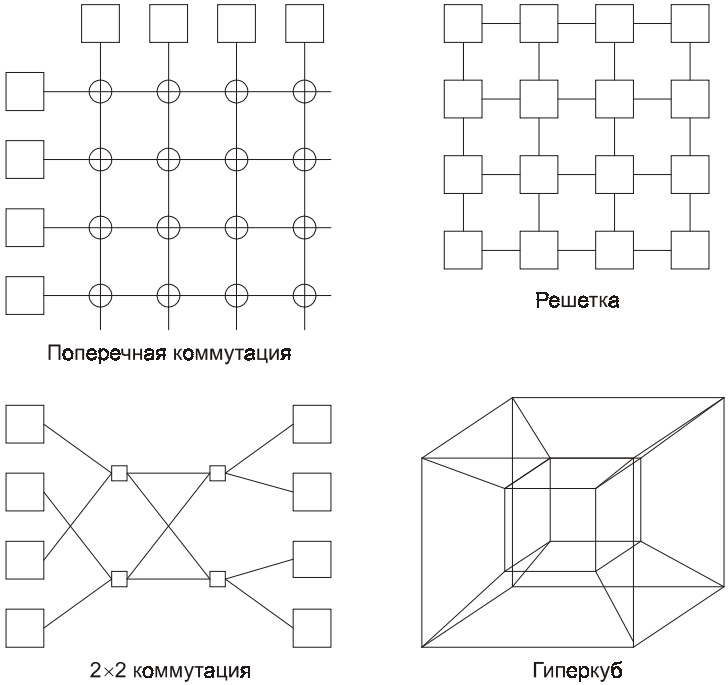


Рис. 6.14. Некоторые типы составных коммутаторов

6.5.3. Некоторые классификации параллельных архитектур

Кроме классической классификации Флинна существует множество других классификаций: Фенга, Шора, Хендлера, Хокни, Шнайдера, Джонсона, Базу, Кришнамурти, Скилликорна, Дазгупты, Дункана (**http://www.parallel.ru**). Рассмотрим подробнее некоторые из них.

6.5.3.1. Классификация Хокни

Классификация Хокни позволяет далее систематизировать архитектуры, относящиеся к очень широкому классу MIMD по классификации Флинна. Основная идея состоит в том, что множественный поток команд может быть обработан двумя способами.

- Одно конвейерное устройство обработки работает в режиме разделения времени для отдельных потоков. Такие компьютеры называют *конвейерными*.
- Каждый поток обрабатывается своим собственным устройством. Такие вычислительные системы в свою очередь делятся на два класса.
  - Компьютеры, в которых возможна прямая связь каждого процессора с каждым процессором, реализуемая с помощью переключа-

теля. Среди них выделяют компьютеры с общей и распределенной памятью.

- Компьютеры, в которых прямая связь каждого процессора возможна только с ближайшими соседями по сети, а взаимодействие удаленных процессоров поддерживается специальной системой маршрутизации через процессоры-посредники. Считается, что все такие компьютеры имеют распределенную память, а дальнейшая классификация определяется топологией сети: иерархические структуры, гиперкубы, регулярные решетки и т. п.

6.5.3.2. Классификация Кришнамурти

В данной классификации предлагается использовать четыре качественные характеристики параллелизма.

- Степень гранулярности, определяющей уровень параллелизма. Классификация выделяет параллелизм на уровне данных, команд и задач.
- Способ реализации параллелизма, который может быть либо чисто аппаратным, либо аппаратно-программным.
- Топология и природа связи процессоров. Топология связи близка к способу реализации параллелизма. Для аппаратного способа следует рассматривать топологию связей процессора (например, кольцо, звезда и т. п.) и степень связности процессоров между собой (сильная, слабая, средняя). Для аппаратно-программной реализации следует дополнительно учесть механизм реализации связи: передача сообщений, общая память и т. п.
- Способ управления процессорами, определяющий общий принцип функционирования всей совокупности процессоров вычислительной системы: синхронный, управляемый потоками данных, или асинхронный.

Принцип построения классификации иерархический. Для каждой из степеней гранулярности рассматриваются все способы реализации параллелизма и т. п. (рис. 6.15).

На основе четырех выделенных характеристик можно определить место наиболее известных классов архитектур в данной классификации.

Например, для параллельных векторных систем:

- степень гранулярности — на уровне данных;
- способ реализации параллелизма — аппаратный;
- топология — простая, степень связности процессоров — средняя;
- способ управления процессорами — синхронный.

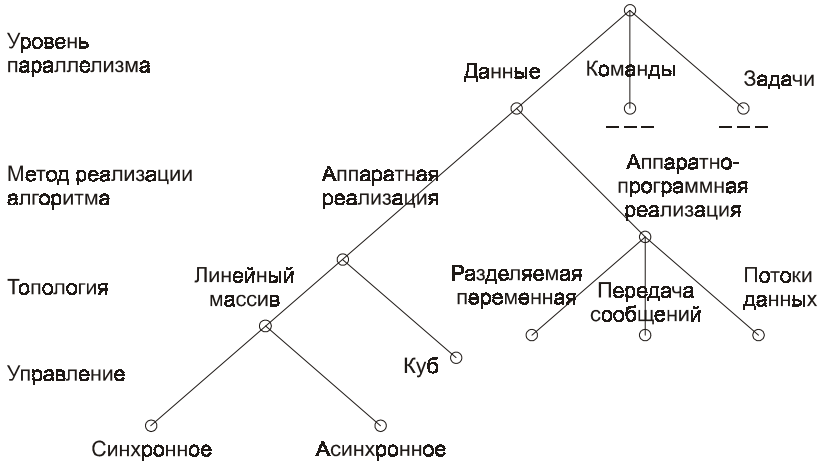


Рис. 6.15. Классификация параллельных вычислительных систем

6.6. Архитектура компьютерных сетей

Вообще говоря, компьютерная сеть — частный случай распределенной архитектуры. Однако это очень важный случай, которому мы посвятим данный раздел.

6.6.1. Классификация сетей и сетевые топологии

6.6.1.1. Классификация сетей по типу технологии передачи данных

По технологии передачи данных сети можно разделить на два основных типа.

- Тип *вещание* (от одного ко многим), имеющий режим, когда сообщение, отправленное одной машиной, получают все машины в сети. В определенном поле сообщения указан адрес получателя. Будет ли обработан пакет на данной машине (т. е. передан операционной системе), зависит от ряда факторов.
- Тип *точка-точка*, соединяющий каждую пару машин индивидуальным каналом.

6.6.1.2. Классификация сетей по сетевым топологиям

Сетевая топология — геометрическая форма (физическая связность) сети. Наиболее распространенными являются топологии шина, кольцо, звезда, иерархия и ячеистая (рис. 6.16).

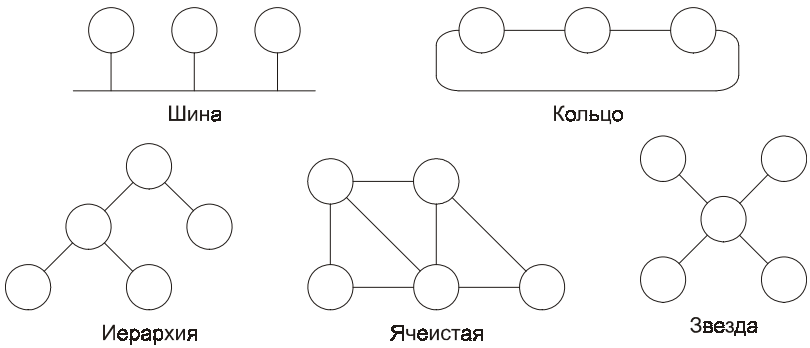


Рис. 6.16. Сетевые топологии

6.6.1.3. Классификация сетей по масштабу

Ниже приведем классификацию сетей по масштабу.

- *Локальные* сети находятся в частном ведении пользователя и расположены на ограниченной территории. Они имеют следующие характеристики:
  - высокоскоростные каналы передачи данных;
  - устройства, соединяемые каналами, находятся недалеко друг от друга;
  - топология сети имеет тенденцию к упорядоченности и структурной однородности. Как правило, она шинная или кольцевая.
- *Городские* сети охватывают несколько зданий в пределах одного города или весь город целиком.
- *Региональные* сети охватывают значительное географическое пространство (например, страны и континенты). Их характеристики таковы:
  - каналы сравнительно медленные, обычно предоставляются телефонными компаниями;
  - устройства, соединяемые каналами, находятся на значительном удалении друг от друга;
  - топология сети характеризуется нерегулярностью. Как правило, она ячеистая или иерархическая.

> ***О каналах передачи данных для региональных сетей***
>
> Обратим внимание на то, что в современном мире утверждение о медленных каналах становится не совсем верным. Появление оптоволоконных каналов существенно увеличивает скорость передачи данных в региональных сетях.

- *Глобальные* сети охватывают всю планету. Примером является сеть Интернет.

6.6.2. Стандарты в области сетей IEEE 802

Институт инженеров по электротехнике и электронике разработал ряд стандартов в области сетей [Бертсекас, Галлагер 1989]. Далее более подробно мы рассмотрим следующие три стандарта:

- 802.3 — шинный доступ;
- 802.4 — маркерная шина;
- 802.5 — маркерное кольцо.

Перечислим еще несколько существующих стандартов:

- 802.1 — общий документ;
- 802.2 — управление логическим каналом;
- 802.6 — городские сети;
- 802.11 — беспроводные сети;
- 802.12 — высокоскоростные компьютерные сети.

6.6.2.1. Метод доступа с контролем несущей и обнаружением коллизий

Доступ с контролем несущей и обнаружением коллизий (Carrier Sense Multiple Access/Collision Detection — CSMA/CD) является методом доступа с состязанием [Блэк 1990].

Каждый узел может в любой момент начать передачу пакета. *Коллизия* — сталкивание пакетов при одновременной передаче. Узлы сами обнаруживают коллизии. В случае возникновения коллизии оба узла вводят задержку посылки пакетов случайным образом, а затем осуществляют повторную посылку пакетов.

Реальное воплощение стандарта на метод доступа с контролем несущей и обнаружением коллизий — это протокол Ethernet. Существуют незначительные различия в формате кадра между Ethernet и IEEE 802.3.

6.6.2.2. Маркерная шина

В случае маркерной шины право использования среды передается от узла к узлу не состязательным путем, а организационным способом с помощью маркера. Маркер перемещается по логическому кольцу физической шинной топологии с использованием адресации узлов. В случае появления нового узла, передача пакетов приостанавливается, и система устанавливает новый порядок обхода логического кольца.

6.6.2.3. Маркерное кольцо

Маркерное кольцо представляет собой топологию физического кольца, имеющего узлы и однонаправленные звенья. Для обеспечения доступа к сети на основе приоритетов используется маркер. Маркер может находиться в двух состояниях: свободном и занятом. Свободный маркер передается от узла к узлу, если ни у одного узла нет пакетов для передачи. Если у узла есть пакет для передачи, то он должен ждать прихода свободного маркера. Далее он изменяет состояние маркера на занятое и передает его далее по кольцу с пакетом данных. Узлы регенерируют сигнал, проверяют адрес и передают данные следующей станции. При прохождении через узел назначения пакет копируется. В конце концов, данные будут получены станцией-отправителем, которая изменяет состояние маркера на свободное и передает его следующему узлу.

Одна из станций — *сетевой активный монитор* — порождает новые маркеры, если это необходимо, и генерирует диагностические кадры. Монитор выбирается при инициализации кольца, и им может быть любая машина.

При выходе из строя хотя бы одного узла возникают проблемы целостности кольца. Для решения этих проблем возможно введение дополнительных связей между узлами.

Протокол, который поддерживает маркерное кольцо, — Token Ring. Он используется реже, чем Ethernet из-за достаточно дорогого оборудования. Однако скорость передачи данных этого протокола более высокая.

6.6.3. Аппаратная поддержка локальных сетей

Рассмотрим аппаратную поддержку локальных сетей на примере сети Ethernet. Известны следующие среды передачи, используемые в Ethernet:

- на толстом коаксиальном кабеле (10Base-5);
- на тонком коаксиальном кабеле (10Base-2);
- на витой паре (10Base-T);
- на волоконно-оптической линии (10Base-F).

В исходной спецификации Ethernet использовался толстый коаксиальный кабель, далее появился тонкий коаксиальный кабель. В 90-х годах XX века наиболее распространенной средой стала витая пара. Постоянная величина для всех перечисленных сред — временное окно Ethernet (51,2 микросекунды). Сигнал, посланный компьютером на одном конце сети, должен достигнуть компьютера на другом конце не более чем за половину этого времени.

6.6.3.1. Ethernet на толстом коаксиальном кабеле (10Base-5)

Пример данной локальной сети приведен на рис. 6.17.

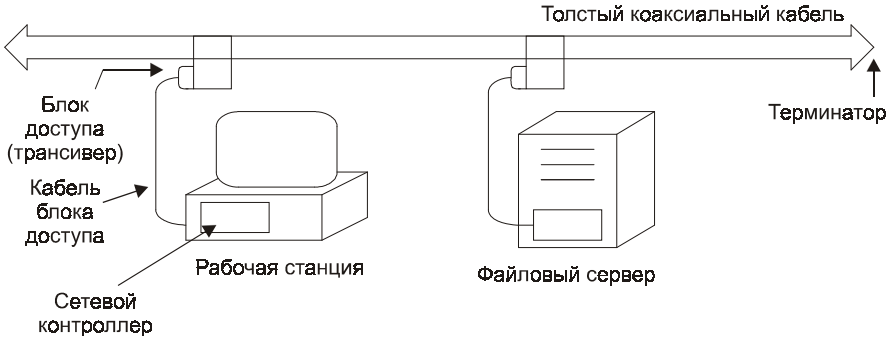


Рис. 6.17. Ethernet на толстом коаксиальном кабеле

Уточним некоторые понятия, приведенные на рисунке.

- *Сетевой контроллер (сетевой адаптер, сетевая плата)* — это печатная плата, установленная в каждом компьютере. Контроллер реализует функции физического уровня и управление доступом к передающей среде.
- *Блок доступа (трансивер)* — устройство для подключения кабеля блока доступа к толстому коаксиальному кабелю.
- *Терминатор* — разъем, устанавливаемый в качестве заглушки на концах сегмента кабеля.

Существуют некоторые ограничения на тактико-технические характеристики данной сети. Они зависят от изготовителя, но их средние значения таковы:

- максимальная длина сегмента кабеля — 500 метров. Реальная длина зависит от скорости передачи и должна гарантировать обнаружение коллизий;
- максимальное число узлов в сети — 100;
- минимальное расстояние между блоками доступа — 2,5 метра.

6.6.3.2. Ethernet на тонком коаксиальном кабеле (10Base-2)

Пример данной локальной сети приведен на рис. 6.18.

Уточним еще одно новое понятие, приведенное на рисунке.

Т-разветвитель — разветвитель, имеющий разъем для подключения к сетевому контроллеру и два гнезда для подключения кабелей от соседних узлов.

Ограничения на тактико-технические характеристики данной сети таковы:

- максимальная длина сегмента кабеля — 185 метров;
- максимальное число узлов в сети — 30;
- минимальное расстояние между блоками доступа — 0,5 метра.

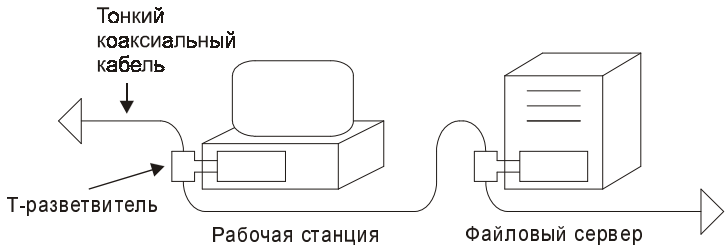


Рис. 6.18. Ethernet на тонком коаксиальном кабеле

6.6.3.3. Ethernet на витой паре (10Base-T)

Пример данной локальной сети приведен на рис. 6.19.

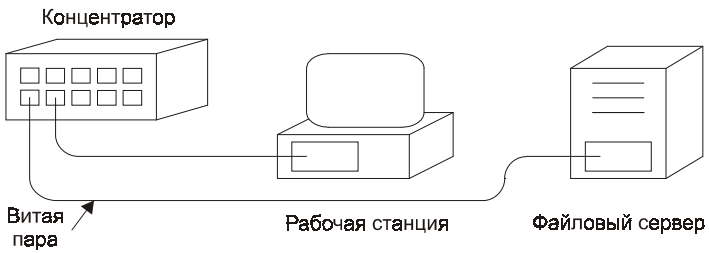


Рис. 6.19. Ethernet на витой паре

Эта быстрая, дешевая, легко перестраиваемая архитектура.

Уточним новое понятие, приведенное на рисунке.

Концентратор (хаб) — устройство согласования и усиления электрических сигналов.

Существует ограничение на длину луча витой пары — 100 метров.

6.6.3.4. Ethernet на волоконно-оптической линии (10Base-F)

Волоконно-оптическая линия обычно применяется в тех случаях, когда расстояние между узлами сети превышает возможности традиционных сред передачи. Волоконно-оптические соединения можно использовать на линиях до двух километров.

6.6.4. Глобальная сеть Интернет

Революционным является то, что обновляет самое себя.
Л. Витгенштейн

На рубеже веков был проведен опрос профессиональных программистов, что оказало на них наибольшее воздействие в последнем десятилетии XX века. Итоги опроса нетрудно предсказать — более половины ответили "Интернет".

6.6.4.1. Введение в Интернет

Интернет-технологии — технологии, позволяющие наступать на грабли, находящиеся на другой стороне земного шара.

Программистский фольклор

Интернет — это имя, под которым объединена группа мировых информационных ресурсов, в том числе совокупность нескольких международных глобальных сетей.

Все официальные стандарты технологии коммуникации в сети Интернет *(интернет-технологии)* публикуются как документы Request for Comments (RFCs) (**http://www.rfc-editor.org/**). Эти документы имеют форму заметок о сети Интернет, в которых обсуждаются многочисленные аспекты коммуникации, включая сетевые протоколы, процедуры, программы и концепции.

С точки зрения программиста, кроме внешней, видимой всем пользователям стороны Интернета существует и реализационная сторона — в виде программной поддержки сети *(см. разд. 7.2.3)*.

Интернет (как сеть, объединяющая ресурсы) обладает двумя основными свойствами.

- *Предоставляет информационные ресурсы*. С точки зрения программиста, в качестве ресурсов могут рассматриваться и библиотеки программ, и инструментарий (например, компиляторы, текстовые редакторы и отладчики).
- *Имеет средства работы с этими ресурсами*. Перечислим некоторые из таких средств: электронная почта, программа telnet, программа ftp, телеконференции, среда WWW, программа chat, интернет-браузеры.

6.6.4.2. История и эволюция Интернета

Начало истории Интернета было положено созданием в США научной структуры — агентства ARPA (Advanced Research Projects Agency). Приведем некоторые даты, связанные с историей Интернета.

- 1967 г. — в Англии была создана первая в мире локальная вычислительная сеть. Состоялась презентация сети для ARPA.
- 1968 г. — создан первый пакетный коммутатор для ARPANET (ARPA NETwork). Интерфейсные пакеты сообщений (Interface Message Processor — IMP) выполняли функции, подобные функциям современных маршрутизаторов. Это была разработка компании Bolt, Beranek & Newman.
- 1969 г. — в сентябре состоялась передача первого компьютерного сообщения между узлами в Калифорнийском университете (Лос-Анжелес) и Стэнфордском исследовательском институте. Эту дату многие считают днем рождения Интернета.
- 1970 г. — в ARPANET применяют Network Control Protocol. Появление протокола ALOHA — прообраза Ethernet.

- 1971 г. — ARPANET насчитывает 15 узлов.
- 1971 г. — Рэй Томлисон (Ray Tomlison) из компании Bolt, Beranek & Newman послал первое сообщение по электронной почте.
- 1978 г. — TCP/IP используется в других сетях (например, PRNET и SATNET). Разнообразие технологий пакетной коммутации.
- 1 января 1983 г. — переход ARPANET на протоколы TCP/IP. Эта дата также претендует на день рождения Интернета.
- 30 апреля 1995 г. — движение информации по Интернету было передано в ведение коммерческих сетей.

Отметим также некоторые даты, связанные с началом Российской истории Интернета:

- Июль 1989 г. — проведен первый обмен сообщениями с Финляндией (**http://news.demos.su/private/demos.html**). Выполнили Вадим Антонов, Дмитрий Володин (Москва), Лео Томберг (Таллин), Петри Ояла (Хельсинки).
- 1990 г. — образование сети Relcom. Первоначально она объединяла компьютеры ИВЦ ИАЭ им. И. В. Курчатова и кооператива Демос (Москва). В августе установлена устойчивая связь с европейской сетью EUnet.
- 20 июля 1991 г. — образование сети FREEnet. Корпоративная, некоммерческая компьютерная сеть основана по инициативе Института органической химии им. Н. Д. Зелинского (Москва).

В 1998 году американский журнал Forbes под заголовком "Легенды Сети" опубликовал имена самых влиятельных людей, внесших огромный вклад в развитие сети Интернет:

1. Линус Торвальдс (Linus Torvalds) (28 лет) — создатель операционной системы Linux.
2. Ричард Столлман (Richard Stallman) (45 лет) — основатель Фонда свободного программного обеспечения (Free Software Foundation).
3. Тим Бернерс-Ли (Tim Berners-Lee) (43 года) — разработчик World Wide Web.
4. Роб Глейзер (Rob Glaser) (36 лет) — основатель компании RealNetworks.
5. Джерри Янг (Jerry Yang) (29 лет) — основатель портала Yahoo!

Не случайно первое место отведено Торвальдсу. Благодаря доступности в паре с сервером Apache (**http://www.apache.org**) операционная система Linux прочно заняла место в сети Интернет.

В табл. 6.12 отражен рост количества инсталляций систем, связанных с Интернетом.

Таблица 6.12. *Количество инсталляций систем, связанных с Интернетом*

Система	Инсталляции (1996 год)	Инсталляции (2000 год)
Серверы Интернета	15 000 000	75 000 000
Компьютеры с доступом в Интернет	45 000 000	300 000 000
Компьютеры с электронной почтой	55 000 000	400 000 000

Книг, посвященных сети Интернет, выпущено огромное количество. Пожалуй, одними из самых интересных и оригинальных остаются (несмотря на их достаточно почтенный возраст) следующие руководства:

- "Все об Internet" [Крол 1995];
- "Секреты Internet" [Левин, Бароди 1996];
- "Желтые страницы Internet (русские ресурсы)" [Сигалов 1996]. Можно считать, что каталог русскоязычного Интернета начинался именно с этой работы.

Кстати, очень логично будет предложить читателю найти информацию об Интернете в самом Интернете. Последовательность действий проста. Следует зайти в сетевом браузере на один из поисковых серверов. Далее нужно ввести текстовую строку, например "информационные ресурсы Интернета", и выполнить запрос на поиск информации.

Миллионы программистов благодарны сети Интернет только за то, что именно ее развитие нарушило господство компании Microsoft в индустрии программистской инженерии и производства программных продуктов. Именно в середине 90-х годов XX века мы узнали о технологиях и программных продуктах компаний Netscape, Sun Microsystems, Inc. и многих других. Сколь бедно было бы наше представление о программировании, если этого бы не произошло.

В данной главе мы рассматривали сети (в том числе, Интернет) с аппаратной точки зрения. В следующей главе мы обратимся к сетям с точки зрения программной поддержки со стороны операционных систем.

6.7. Заключение

6.7.1. Выводы и рекомендации

Компания Intel после выпуска чипов Pentium I, Pentium II, Pentium III и Pentium IV решила выпустить чипы "Pentium возвращается" и "Pentium жив"

Программистский фольклор

ЭВМ (компьютер) — основа вычислений. Без реальной ЭВМ современные вычислительные дисциплины были бы всего лишь ветвью теоретической

математики. Профессиональный программист не должен воспринимать ЭВМ как черный ящик, исполняющий программы с помощью неизвестной волшебной магии. Следует понимать устройство функциональных компонент, из которых состоит компьютер, их характеристики, производительность и взаимодействие между ними. Знание архитектуры и организации ЭВМ также позволяет писать более эффективные программы. При выборе архитектуры важно понимать значение характеристик различных компонентов, таких как тактовая частота процессора и объем памяти.

Современный этап мирового развития компьютерной техники характеризуется переходом на высокопроизводительные вычислительные системы высокого параллелизма. Открываются новые возможности для решения фундаментальных научных проблем, многие из которых имеют практическое значение [Бурцев 2003].

- Прогноз погоды, экологических изменений и природных геофизических явлений.
- Исследование структуры сложных газодинамических и гидродинамических течений.
- Изучение свойств веществ в интересах атомной энергетики.
- Структурное исследование человеческих генов.
- Молекулярное конструирование лекарств.
- Проектирование сверхсложных радиоэлектронных комплексов и т. д.

Заметим, что на математическом уровне задача распараллеливания вычислительного процесса успешно решена, поскольку большинство физических задач сводится к решению уравнений математической физики. Однако какова бы ни была архитектура ЭВМ, программисту приходится решать две практические задачи [Бурцев 2003]:

- распределить вычислительные процессы по процессорам или машинам так, чтобы они были равномерно загружены;
- обеспечить синхронизацию вычислительного процесса так, чтобы данные одного процессора поступали на вход другого в соответствии с алгоритмом, определенным задачей.

Поскольку самому программисту решить их практически нереально, решение этой задачи должно быть переложено на аппаратуру.

Объединяющей основой функционирования и поэтапного развития систем являются концепции и стандарты открытых систем (масштабируемость, портабельность и т. п.). Этот подход за последние 2—3 года в мировой практике выражен в развитии кластерных вычислительных систем. Такие системы строятся на базе современных массовых микропроцессоров Pentium IV, Xeon (**http://www.intel.com**), Power 4 (**http://www.ibm.com**), Opteron (**http://www.amd.com**).

В 1993 году стартовал проект TOP500 (**http://www.top500.org**). Он был задуман как единое место для отслеживания и определения тенденций в высокопроизводительных вычислениях. Дважды в год на сайте обновляется список, состоящий из 500 наиболее мощных вычислительных систем по всему миру. Производительность систем определяется на основе пакета тестов Linpack для решения систем линейных алгебраических уравнений (**http://www.top500.org/lists/linpack.php**). Список содержит подробную информацию о спецификации систем и областях их применения.

Серьезные изменения происходят в настоящее время в стиле общения с компьютером. Стал привычным графический ввод информации, распознавание рукописного текста, растет интерес к голосовому вводу. Сам компьютер внешне также претерпевает изменения. Пример здесь подает компания Apple, давно пришедшая к выводу, что кроме собственно технологических возможностей компьютеров очень важен их внешний вид, в частности цвет.

6.7.2. Проблемы и перспективы развития

Опасность не в том, что машина начинает уподобляться человеку,
а в том, что человек превращается в подобие машины.
Сидней Харрис

С функциональной точки зрения происходит усиление разделения компьютеров на две группы: клиенты и серверы. По материалам журнала IEEE Spectrum [Computers 1997], к началу XXI века существенными будут сегменты компьютеров, изображенные на рис. 6.20.

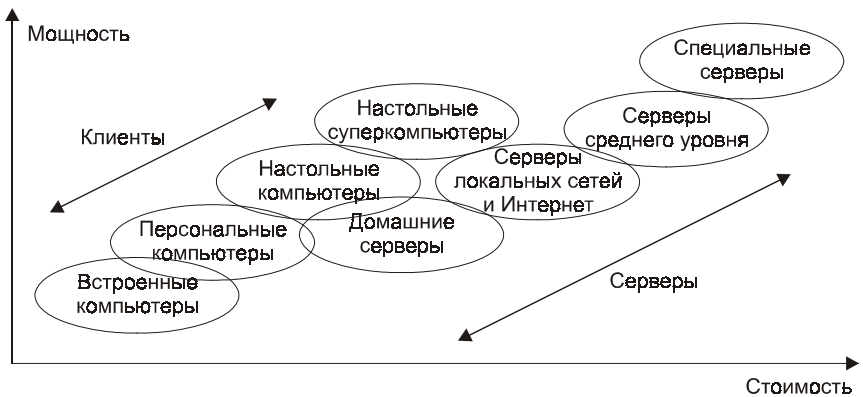


Рис. 6.20. Функциональные классы компьютеров (прогноз на начало XXI века)

С точки зрения создания качественно новых высокопроизводительных вычислительных систем, перспективными являются три направления.

- *Квантовые компьютеры* — устройства, основной строительной единицей которых является кубит (qubit, quantum bit). Для описания состояний

квантовой системы было введено понятие волновой функции, значение которой представляется в виде вектора с большим числом значений. Квантовая система может находиться в состоянии с волновой функцией, равной линейной комбинации собственных функций, соответствующих каждому из возможных значений. Это означает, что кубит в одну единицу времени равен и нулю, и единице. Классический бит в единицу времени может быть равен либо нулю, либо единице. За счет того, что квантовые вычисления обладают свойством естественного параллелизма, квантовому компьютеру необходимо гораздо меньше времени для решения задач определенного класса.

- *Нейрокомпьютеры* (биокомпьютеры) — устройства, создаваемые на базе нейрочипов, ориентированных на решение конкретной задачи. Создание таких компьютеров базируется на теории перцептронов — искусственных нейронных сетей, которые могут обучаться распознаванию образов. Компьютеры, основанные на структурах, имеющих свойства мозга и нервной системы, помогают при решении сложных задач параллельной обработки информации. Они имеют свойства самообучаемости и высокой надежности.
- *Оптические компьютеры* — устройства обработки информации с использованием света. Свет — электромагнитная волна, имеющая частоту, превосходящую в 10—100 миллионов раз частоту волны электрической. С ее помощью в некоторый интервал времени можно передавать большее число сигналов, а следовательно, и информации. Важнейшая особенность оптических компьютеров — аналоговый характер информации, с которой они работают. Тем самым можно эффективно работать с компьютерными преобразованиями, отличными от цифровых (например, преобразованиями Фурье).

С проблемами и перспективами развития классических аппаратных средств тесно связаны два закона.

- *Закон Мура* утверждает, что удвоение числа транзисторов в микросхеме происходит каждые 1,5—2 года. Эту закономерность Гордон Мур (Gordon Moore) обнаружил в 1965 году, представив в виде графика рост производительности запоминающих микросхем. Новые модели микросхем разрабатывались спустя одинаковые периоды — 18—24 месяца после появления их предшественников. Их емкость при этом возрастала примерно вдвое. Закон объясняет, почему устаревание компьютеров происходит так стремительно — всего за несколько лет. Некоторые специалисты считают, что действие этого закона продлится до 2015 года.
- Перед тем как дать формулировку следующего закона, предположим, что доля операций, которые должны быть реализованы в процессе выполнения программы последовательно, равна величине `f` (где $0 \le f \le 1$).

Крайние значения соответствуют полностью параллельным (`f = 0`) и полностью последовательным программам (`f = 1`). *Закон Амдала* утверждает, что для того чтобы оценить, какое ускорение `S` может быть получено на вычислительной системе из `p` процессоров при заданной величине `f`, следует воспользоваться формулой `S ≤ 1/(f + (1 - f)/p)`. Некоторые специалисты образно называют этот закон "страшным", указывая на следующий факт. Предположим, что в программе всего лишь 10% последовательных операций (`f = 0.1`). Закон утверждает, что, сколько бы процессоров не было использовано, максимальное ускорение работы может быть не более чем в 10 раз. Причем 10 получается только в том случае, если время исполнения параллельной части равно нулю.

6.7.3. Рекомендации по литературе

Обратим особое внимание на следующие материалы, посвященные компьютерным архитектурам.

- "Архитектура компьютера" [Таненбаум 2002]. Книга посвящена структурной организации компьютера, рассматривающегося как иерархия уровней, включая цифровой логический уровень и уровень архитектуры команд.
- "Архитектура современных ЭВМ" [Майерс 1985]. Книга посвящена анализу фундаментальных проблем архитектуры.
- "Компьютеры: справочное руководство" [Компьютеры 1986]. В 1-м томе этого трехтомника рассматриваются аппаратные средства ЭВМ.
- "Код" [Петцольд 2001]. В книге подробно излагаются принципы работы процессора и других устройств компьютера.
- "Структуры ЭВМ и их математическое обеспечение" [Королев 1978]. Работа содержит обзор отечественных и зарубежных универсальных вычислительных машин.

Параллельным вычислительным архитектурам посвящены следующие источники.

- "Параллельные вычисления" [Воеводин, Воеводин 2002]. Книга известных российских ученых посвящена обсуждению ключевых проблем современных параллельных вычислений.
- "Параллельные вычислительные системы" [Корнеев 1999]. Работа дает комплексное представление об аппаратных средствах массово параллельных вычислительных систем.
- В Интернете существует русскоязычный сервер, посвященный параллельным архитектурам (**http://www.parallel.ru**).

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Архитектура и организация ЭВМ". Основные разделы этой темы следующие:
  - цифровая логика и цифровые системы;
  - организация машины на уровне ассемблера;
  - устройство памяти компьютера;
  - функциональная организация;
  - многопроцессорные и альтернативные архитектуры;
  - архитектура сетевых и распределенных систем.
- Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет дисциплину специализации "Архитектура вычислительных систем и компьютерных сетей".

6.7.4. Вопросы и задания

1. Опишите эволюцию компьютерных архитектур от электронных ламп до СБИС.
2. Объясните причины использования различных форматов представления числовых данных.
3. Как представление данных с фиксированной разрядностью влияет на точность?
4. Объясните организацию классической машины фон Неймана и ее основных функциональных блоков. Как происходит выполнение инструкции в машине фон Неймана?
5. Объясните, как инструкции представляются на машинном уровне и на уровне языка Assembler.
6. Объясните особенности различных форматов инструкций, в частности, форматов с фиксированной и переменной длинами.
7. Напишите небольшие фрагменты ассемблерной программы.
8. Продемонстрируйте реализацию фундаментальных высокоуровневых конструкций программирования на машинном уровне.
9. Объясните базовые понятия прерываний и операций ввода-вывода.
10. Перечислите основные виды памяти. Объясните основы управления памятью. Объясните роль кэша.
11. Обсудите понятие параллельной обработки в моделях, отличающихся от модели фон Неймана.

12. Опишите альтернативные архитектуры, такие как SIMD, MIMD и VLIW.
13. Опишите суперскалярные архитектуры и их преимущества. Объясните понятие предсказания переходов и его значение.
14. Опишите связь масштабируемости и производительности.
15. Объясните основные компоненты сетевых систем и различия между локальными и глобальными сетями.
16. Обсудите архитектурные вопросы, связанные с сетевыми вычислениями и распределенными мультимедиа-технологиями.
17. Объясните основы параллелизма на уровне инструкций с использованием конвейера и основные проблемы, которые могут возникнуть при этом.

ЛИТЕРАТУРА

Мысли не сохраняются, их надо во что-то воплотить.

А. Уайтхед

1. *Бертсекас, Галлагер 1989* — Бертсекас Д., Галлагер Р. Сети передачи данных. — М.: Мир, 1989.
2. *Блэк 1990* — Блэк Ю. Сети ЭВМ: протоколы, стандарты, интерфейсы. — М.: Мир, 1990.
3. *Векторизация 1991* — Векторизация программ: теория, методы, реализация. Сборник статей. — М.: Мир, 1991.
4. *Воеводин, Воеводин 2002* — Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002.
5. *Компьютеры 1986* — Компьютеры: справочное руководство. В 3-х томах. — М.: Мир, 1986.
6. *Корнеев 1999* — Корнеев В. В. Параллельные вычислительные системы. — М.: "Нолидж", 1999.
7. *Королев 1978* — Королев Л. Н. Структуры ЭВМ и их математическое обеспечение. — М.: Наука, 1978.
8. *Крол 1995* — Эд Крол. Все об Internet. — К.: Торгово-издательское бюро BHV, 1995.
9. *Левин, Бароди 1996* — Левин Джон Р., Бароди Кэрол. Секреты Internet. — К.: "Диалектика", 1996.
10. *Майерс 1985* — Майерс Г. Архитектура современных ЭВМ. В 2-х кн. — М.: Мир, 1985.
11. *Петцольд 2001* — Петцольд Ч. Код. — М.: Издательско-торговый дом "Русская Редакция", 2001.

12. *Райс 1990* — Райс Л. Эксперименты с локальными сетями микроЭВМ. — М.: Мир, 1990.

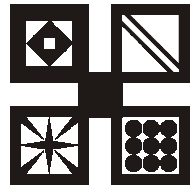
13. *Сигалов 1996* — Сигалов А. Желтые страницы Internet (русские ресурсы). — СПб.: Питер, 1996.

14. *Таненбаум 2002* — Таненбаум Э. Архитектура компьютера. — СПб.: Питер, 2002.

15. *Хемминг 1972* — Хемминг Р. В. Численные методы. — М., 1972.

16. *Paul 1994* — Richard P. Paul. SPARC architecture, assembly language programming, and C. Prentice-Hall, Inc. 1994.

17. *SPARC 1994* — The SPARC Architecture Manual. Prentice-Hall, Inc. 1994.



Глава 7

Операционная платформа

Изучение механизма и структуры операционных систем необходимо по следующим причинам:

- основные идеи, концепции и алгоритмы, лежащие в основе операционных систем, применимы ко многим другим областям программирования, и особенно к системному программированию;
- операционная система — большая и очень сложная программа, на примере которой можно изучать вопросы создания сложных программных продуктов;
- такие популярные программные продукты, как системы управления базами данных, могут рассматриваться как надстройки над операционными системами.

7.1. Введение в операционные системы

7.1.1. Основные понятия и определения

Операционная система (ОС) — это часть программного обеспечения, выступающая в качестве интерфейса между приложениями (и пользователями) и аппаратурой компьютера. Операционная система выполняет две основные функции [Олифер, Олифер 2001].

- *Предоставление пользователю-программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины*, с которой удобнее работать. Виртуальная машина — это вычислительная система заданной конфигурации, моделируемая для пользователя программными и аппаратными средствами конкретного реально существующего компьютера. Операционная система является тем слоем программного обеспечения, который преобразует аппаратную машину в виртуальную. Конфигурация виртуальной машины может существенно отличаться от реальной.

- *Повышение эффективности использования компьютера за счет рационального управления его ресурсами.* Ресурсы операционной системы можно разделить на две группы (рис. 7.1):
  - *программные ресурсы* (процессы, виртуальное адресное пространство, подсистема ввода-вывода);
  - *аппаратные ресурсы* (процессоры, память, устройства).

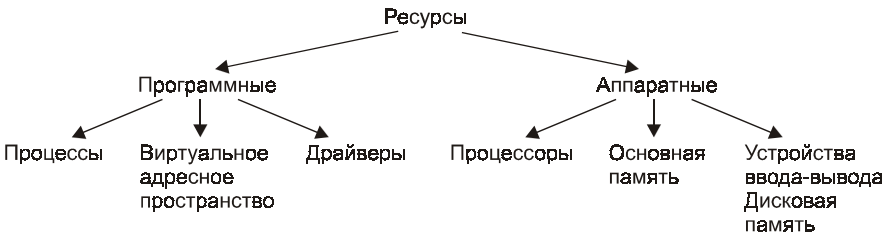


Рис. 7.1. Классификация ресурсов

Обратим внимание на то, что практически каждому аппаратному ресурсу соответствует некоторый программный ресурс, тесно с ним связанный (например, процессор и процесс).

Об основной функции операционной системы

Образно говоря, основной функцией операционной системы можно считать чародейство — превращение системы в нечто большее, чем есть на самом деле. Например, операционная система может создать иллюзию одновременного исполнения нескольких программ на одном процессоре. В итоге пользователь воспринимает виртуальную машину как компьютер, имеющий архитектуру, отличную от реально существующей.

Ядро операционной системы — модули, выполняющие основные функции операционной системы. Эти модули обычно поддерживают управление процессами, памятью, устройствами ввода-вывода. Код ядра операционной системы исполняется в привилегированном режиме работы процессора.

Некоторые компоненты операционной системы представляют собой обычные приложения в стандартном для данной операционной системы формате. Их называют *вспомогательными модулями операционной системы.* Поэтому часто бывает сложно провести границу между операционной системой и приложениями. Обычно решение о принадлежности некоторой программы операционной системе принимает производитель. Многие компоненты систем программирования, рассмотренные в *гл. 5*, часто вводятся производителями ОС в качестве ее вспомогательных модулей.

7.1.2. История и эволюция операционных систем

7.1.2.1. Поколения операционных систем

Принято выделять следующие исторические поколения операционных систем.

- Нулевое поколение. В первых компьютерах операционные системы отсутствовали. Это период с момента появления первых компьютеров до середины 50-х годов XX века.
- Первое поколение. Пакетная обработка, мультипрограммные операционные системы. Появились в середине 50-х годов XX века.
- Второе поколение. Многорежимные операционные системы, операционные системы реального времени. Появились в середине 60-х годов XX века.
- Третье поколение. Операционные системы для персональных компьютеров, сетевые операционные системы. Появились в начале 80-х годов XX века.
- Четвертое поколение. Распределенные операционные системы. Появились в начале 90-х годов XX века.

7.1.2.2. Краткий обзор истории создания операционных систем

```
int main() { printf(&unix["\021%six\012\0"], (unix) ["have"]+"fun"-0x60);}
```

Дэвид Корн

Первая операционная система появилась в середине 50-х годов XX века. Она была создана в исследовательской лаборатории компании General Motors для компьютера IBM-701. Практически до середины 60-х годов XX века операционные системы распространялись бесплатно.

В СССР одним из первых подобий операционных систем была система интерпретации обращений к стандартным подпрограммам ИС-2 для М-20 (1959 г.). Она была построена по принципу динамической загрузки библиотечных подпрограмм в область оперативной памяти, выделяемой программистом. Подпрограммы по мере обращения к ним вызывались на рабочее поле, организованное частью как стек (для "фиксируемых" подпрограмм), частью как очередь, откуда они вытеснялись при переполнении поля, но могли возвращаться при новых обращениях.

На рис. 7.2 приведены наиболее известные операционные системы и указаны годы их создания. Дадим краткую характеристику некоторых из них.

- 1961 г. — Atlas. Эта операционная система была разработана в Манчестерском университете (Англия). Многие особенности, впервые появившиеся в ней (например, страничное управление памятью), сейчас являются стандартными частями современных операционных систем.

- 1962 г. — CTSS. Система была разработана в Массачусетском технологическом институте (США) как экспериментальная система с механизмом разделения времени.
- 1965 г. — XDS-940. Система разработана в университете Беркли (США). Это была операционная система с разделением времени. Она давала возможность пользовательской программе определять процессы и работать с ними посредством системных вызовов.

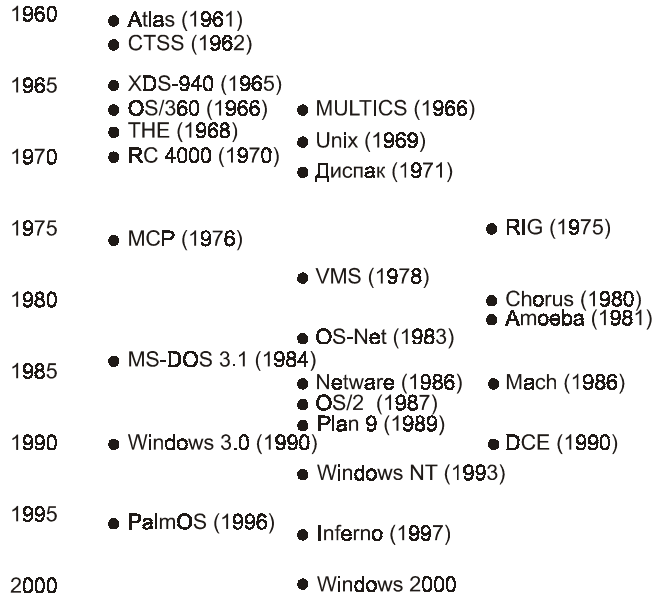


Рис. 7.2. Наиболее известные операционные системы

- 1966 г. — MULTICS. Система также была разработана в Массачусетском технологическом институте и являлась развитием системы CTSS. В свою очередь эта система является предшественницей операционной системы Unix.
- 1966 г. — OS/360. Операционная система для большого семейства компьютеров IBM/360 была написана на ассемблере тысячами программистов и представляла собой миллионы строк кода.

Об ОС ЕС

В 1971 году была выпущена ДОС ЕС 1.0 — полный функциональный отечественный аналог DOS/360 MFT. При переработке текста DOS/360 разработчики сделали большое количество дополнений и улучшений. Система функционировала на семействе ЕС ЭВМ. В результате отечественным пользователям стали доступны прикладные программные продукты, выпущенные за рубежом.

- 1967 г. — Диспетчер-68. Первая операционная система для БЭСМ-6 была создана в ИТМ и ВТ под руководством Л. Н. Королева. Она была

предназначена для управления совместной работой устройств ЭВМ, подготовки и решения задач в мультипрограммном режиме.

- 1968 г. — THE. Система создана в Технической школе Эйндховена (Голландия). Система имела слоистую архитектуру и могла работать с параллельными процессами, выполняя синхронизацию с помощью семафоров.
- 1971 г. — ОС Диспак. Система создавалась В. Ф. Тюриным и группой разработчиков для БЭСМ-6, а затем для МВК Эльбрус.
- 1974 г. — CP/M. Эта система явилась родоначальником семейства операционных систем для микропроцессоров, первоначально для 8-битовых.

Отдельно будем вести разговор об истории и эволюции двух доминирующих в настоящее время семейств операционных систем.

- Семейство Unix. Официальной датой рождения операционной системы Unix считают 1 января 1970 года. С этого момента любая система Unix отсчитывает свое системное время. Реально первая версия этой операционной системы была создана в 1969 году двумя очень талантливыми людьми — Кеном Томпсоном (Ken Thompson) и Деннисом Ритчи (Dennis Ritchie). Истории и эволюции Unix в Интернете посвящено много страниц, например **http://www.levenez.com/unix/**. Существует множество версий, диалектов и клонов Unix, включая System V и BSD — два главных направления развития. Большое количество версий Unix появилось уже в конце 1970-х благодаря тому, что исходный текст системы был широко распространен. Для обеспечения переносимости программ институтом IEEE был разработан стандарт POSIX, определяющий минимальный интерфейс, который должны поддерживать совместимые диалекты Unix. Клон Linux, созданный финским студентом Линусом Торвальдсом, является одним из наиболее популярных среди широкого круга пользователей в настоящее время.

  О том, как правильно писать — UNIX или Unix

  Питер Салюс (Peter H. Salus) [Salus 1994] приводит слова Дагласа Макилроя (Douglas McIlroy) о том, что написание UNIX заглавными буквами является серьезной ошибкой. Мы далее будем придерживаться этой рекомендации и писать Unix.

- Семейство Windows. Первым представителем этого семейства является система MS-DOS 1.0 компании Microsoft. Система поставлялась с персональным компьютером IBM PC, начиная с 1981 года. Версия MS-DOS 3.0 появилась в 1986, а 7.0 — в 1995 году. Добавление графического пользовательского интерфейса к MS-DOS породило в 1985 году систему Windows 1.0. Спустя 10 лет, в 1995 году была выпущена версия Windows 95, далее Windows 98 и, наконец, Windows ME в 2000 году. Все

эти версии имели тесную связь с 16-разрядной MS-DOS, что существенно ограничивало их возможности. Совершенно новая, полностью 32-разрядная операционная система Windows NT 3.1 была выпущена в 1993 году. Массовый переход пользователей на эту систему происходил медленно, но верно. В 1996 году вышла система Windows NT 4.0, Windows 2000 — в 1991, а Windows XP — в 2002 году.

В последние годы получили широкую известность операционные системы для мобильных устройств, например Palm OS, Symbian OS, Nucleus, а также Windows CE.NET.

Существуют даже "учебные" операционные системы, например ОС NACHOS (Not Another Completely Heuristic Operating System), первая версия которой была создана Уэйном Кристофером (Wayne Christopher), Стивом Проктером (Steve Procter) и Томасом Андерсоном (Thomas Anderson) в январе 1992 года. Эта операционная система используется практически во всех университетах мира при выполнении студентами самостоятельных заданий по соответствующему учебному курсу (**http://www.stanford.edu/class/cs140/projects/**).

7.1.3. Классификация операционных систем

7.1.3.1. Классификация по типу централизации

В основу первой и основной классификации положим степень централизации (связности) операционной системы (рис. 7.3).

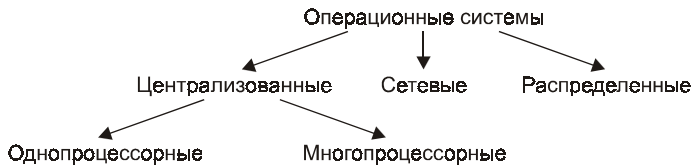


Рис. 7.3. Классификация по типу централизации

Эта классификация принимает во внимание особенности аппаратных платформ, для которых операционные системы создаются.

- *Централизованные* (локальные) операционные системы управляют ресурсами единственного локального компьютера. Они включают два различных с точки зрения алгоритмов подкласса:
  - однопроцессорные системы;
  - многопроцессорные системы.

- *Сетевые* операционные системы предоставляют пользователю сети некоторую виртуальную машину, работать с которой проще, чем с реальной сетевой аппаратурой. Однако пользователь всегда выполняет специальные операции для доступа к сетевым ресурсам. Сетевые системы включают дополнительные компоненты [Олифер, Олифер 2001]:
  - *серверную часть* операционной системы — средства предоставления локальных ресурсов и услуг в общее пользование;
  - *клиентскую часть* операционной системы — средства запроса доступа к удаленным ресурсам и услугам;
  - *транспортные средства* операционной системы — средства обеспечения передачи сообщений между компьютерами сети.
- *Распределенные* операционные системы предоставляют пользователю сети единую централизованную виртуальную машину, которая дает максимальную степень прозрачности сетевых ресурсов. Распределенные системы объединяют все компьютеры сети, для работы в тесной кооперации. При работе в таких системах пользователь, запускающий приложение, не знает, на каком компьютере оно реально выполняется.

Существует интересное обоснование данной классификации [Tanenbaum 1995]. В *гл. 6* было отмечено, что основной характеристикой классификации параллельных и распределенных архитектур считают наличие общей или распределенной (локальной для каждого из узлов) памяти. Исходя из этого, вычислительные системы можно разделить на два класса.

- *Системы с сильными связями.* Сюда относятся системы, состоящие из нескольких однородных процессоров и массива общей памяти.
- *Системы со слабыми связями.* Это системы, состоящие из однородных вычислительных узлов, каждый из которых имеет свою память.

Программное обеспечение также можно разделить на два класса.

- *Программное обеспечение с сильными связями.* Сюда относятся программы, которые при исполнении на нескольких вычислительных модулях в большой степени являются связанными между собой.
- *Программное обеспечение со слабыми связями.* Оно позволяет вычислительным модулям быть независимыми друг от друга, но при необходимости взаимодействовать ограниченным количеством способов.

В результате можно получить четыре различных комбинации между этими парами, три из которых являются осмысленными и определяют следующие типы операционных систем (рис. 7.4).

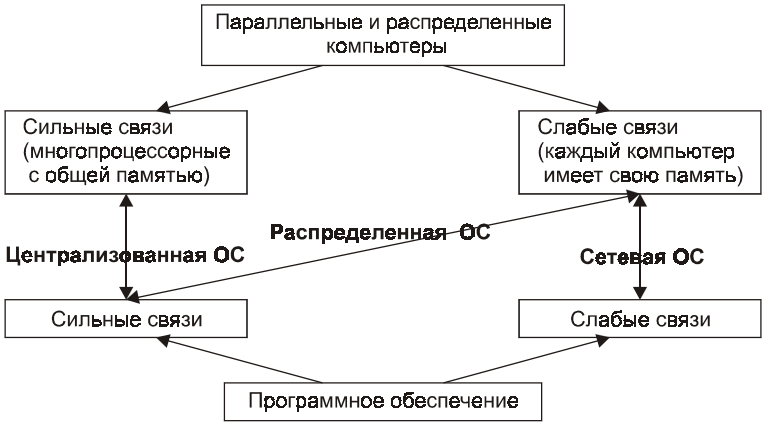


Рис. 7.4. Обоснование классификации по типу централизации

7.1.3.2. Классификация по особенностям алгоритмов управления ресурсами

Классификация ОС по особенностям алгоритмов управления ресурсами имеет следующие аспекты [Олифер, Олифер 2001].

- *Поддержка многозадачности*, а именно:
  - однозадачные операционные системы выполняют функцию предоставления пользователю виртуальной машины, делая простым и удобным процесс его взаимодействия с компьютером;
  - многозадачные операционные системы дополнительно управляют разделением совместно используемых ресурсов. В первую очередь они дают возможность одновременно исполнять несколько задач на одном процессоре.
- *Поддержка многопользовательского режима*, а именно:
  - однопользовательские операционные системы не предоставляют средств защиты информации одного пользователя от несанкционированного доступа другого пользователя. Такие системы не предоставляют возможностей разделения ресурсов;
  - многопользовательские операционные системы такие средства защиты информации имеют.
- *Поддержка многопоточности.* Многопоточные операционные системы дают возможность разделять процессорное время не только между процессами, но и между отдельными ветвями процессов — потоками *(см. разд. 7.2.1.2)*.
- *Поддержка многопроцессорной обработки.* Многопроцессорные операционные системы реализуют более сложные алгоритмы управления ресурсами, предоставляющие возможность работать с несколькими процессорами.

В операционных системах отдельным направлением идет концепция *системы виртуальных машин*. Такие операционные системы допускают одновременную работу нескольких полноценных операционных систем, создавая иллюзию того, что ЭВМ находится в их полном распоряжении.

7.1.3.3. Классификация по особенностям аппаратных платформ

Специфика аппаратных средств, как правило, отражается на специфике операционной системы. В *разд. 6.1.3.1* приведена функциональная классификация компьютеров. Каждый из типов компьютеров в этой классификации имеет определенные свойства, оказывающие непосредственное влияние на свойства операционных систем.

Обратим внимание на то, что наибольший интерес в настоящее время вызывают следующие группы операционных систем:

- операционные системы для мощных серверов;
- операционные системы для рабочих станций и персональных компьютеров;
- операционные системы для карманных компьютеров.

7.1.3.4. Классификация по особенностям областей использования

По особенностям областей использования многозадачные операционные системы могут быть разделены на три типа.

- Операционные системы *пакетной обработки*. Они работают с пакетами задач, причем переключение процессора с одной задачи на другую происходит лишь в том случае, если активная задача сама отказывается от процессора.
- Операционные системы *разделения времени*. Такие системы предоставляют каждой из задач некоторый квант процессорного времени. При этом время ответа программы обычно оказывается достаточно приемлемым, что позволяет использовать эти ОС в качестве диалоговых.
- Операционные системы *реального времени*. Они применяются для управления некоторыми технологическими объектами и процессами. В них существует предельно допустимое время, в течение которого программа должна ответить. *Операционные системы реального времени* характеризуются тем, что в ответ на события они должны гарантированно реагировать *до* определенного времени. Проще говоря, *когда* дается ответ так же важно, как и какой.

Многие современные операционные системы совмещают в себе свойства систем разных типов. Например, часть задач исполняется в режиме разделения времени, а часть — в режиме реального времени.

7.1.3.5. Классификация по типу архитектуры ядра системы

Существуют следующие основные разновидности архитектуры ядра.

- *Монолитное ядро.* Такое ядро компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую (рис. 7.5, *а*).
- *Слоистое ядро.* В этом случае компоненты операционной системы образуют уровни с хорошо продуманной функциональностью и интерфейсом. Как и в предыдущем случае, компоненты работают в привилегированном режиме (рис. 7.5, *б*).
- *Микроядро.* Микроядро выполняет минимум функций по управлению аппаратурой. Обычно в него включаются машинно-зависимые программы, некоторые функции управления процессами и обработка прерываний. Функции более высокого уровня выполняют специализированные компоненты операционной системы: сервер процессора, файловый сервер и т. п. Эти компоненты работают в пользовательском, непривилегированном режиме. Данная архитектура основана на подходе "клиент-сервер" и характеризуется переносимостью, расширяемостью и надежностью (рис. 7.5, *в*).

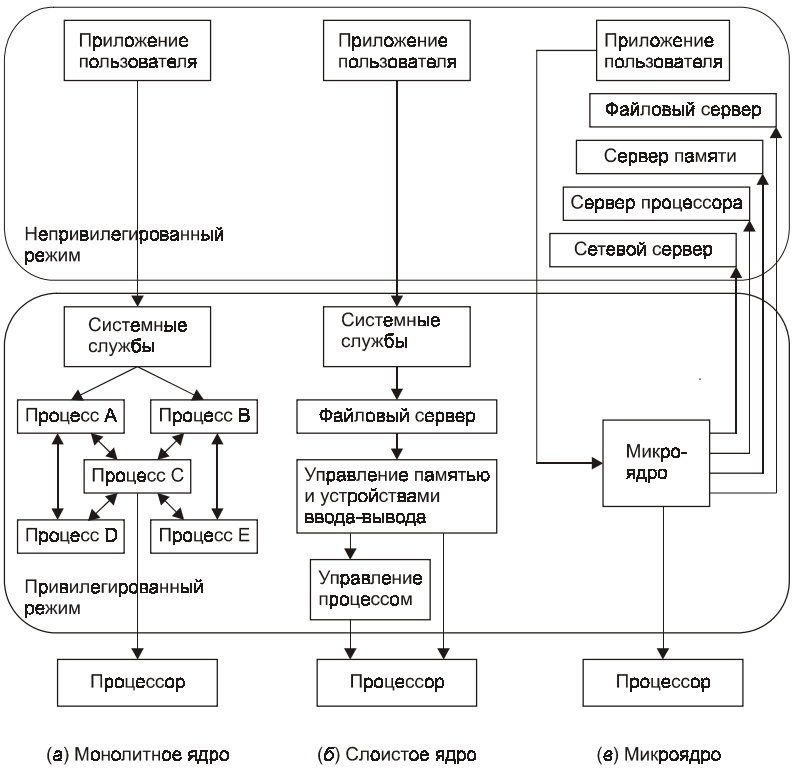


Рис. 7.5. Классификация по типу архитектуры ядра

Концепция системы виртуальных машин может быть поддержана *экзоядром*. Каждый пользователь может быть обеспечен абсолютной копией реального компьютера, но с подмножеством ресурсов. На экзоядро возложена задача распределения ресурсов для виртуальных машин и проверка использования ресурсов.

Отметим, что архитектура ядра в значительной степени влияет на всю архитектуру операционной системы.

7.2. Процессы

Понятие процесса является одним из основных в современных операционных системах. В этом разделе мы подробно рассмотрим сущность процессов и способы их коммуникации и синхронизации в централизованных, сетевых и распределенных операционных системах.

7.2.1. Процессы и потоки (нити) управления

7.2.1.1. Понятие процесса

Мы дадим несколько определений процесса (process), отражающих различные стороны этого важного понятия.

- *Процесс* — это абстракция, описывающая выполняющуюся программу.
- *Процесс* — это исполнение последовательности действий в среде, включающей собственно выполняющуюся программу, а также связанные с ней данные и состояния (открытые файлы, текущий каталог и т. п.).
- С точки зрения операционной системы, *процесс* — это единица работы, заявка на потребление системных ресурсов.
- *Процесс* — объект, которому выделяется процессор.
- *Процесс* — это живая душа программы.

Первое упоминание о процессе появилось в 60-е годы XX века в операционной системе MULTICS.

Если говорить о соотношении между процессом и программой (несмотря на то, что это понятия из разных областей), то справедливы следующие два утверждения:

- программа — это часть состояния процесса. С этой точки зрения процесс — нечто большее, чем просто программа;
- программа может вызывать более чем один процесс для выполнения работы. С этой точки зрения программа — нечто большее, чем процесс.

Процессы образуют иерархию в операционной системе. Соответственно, будем называть порожденные процессы — потомками данного процесса, а родителя порожденного процесса — предком.

Отметим основные состояния процесса (рис. 7.6):

- *на процессоре* — активное состояние, в котором процесс обладает всеми необходимыми ресурсами, в том числе самим процессором;
- *готовность* — процесс находится в очереди на выполнение;
- *ожидание* — процесс ожидает завершения события (например, освобождения ресурса);
- *остановлен* — процесс остановлен, как правило, в отладочном режиме;
- *создание* — выполнение действий, необходимых для создания процесса;
- *завершение* — выполнение действий, связанных с успешным завершением процесса;
- *зомби* — процесс закончен, но предок не принял его завершения.

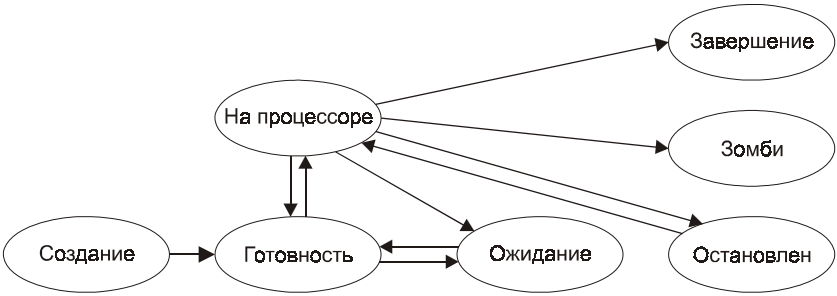


Рис. 7.6. Основные состояния процесса

При создании процесса должны быть выполнены следующие действия:

- присвоение процессу уникального номера (ID);
- добавление процесса в список процессов, известных системе;
- определение начального приоритета;
- формирование блока управления процессом;
- выделение необходимых ресурсов.

В листинге 7.1 приведена простейшая программа на языке C, порождающая новый процесс в ОС Unix.

Листинг 7.1. Порождение процесса в ОС Unix

```
void main() {
    int status;
    int child_pid, child_status;

    child_pid = fork();
```

```
    if ( child_pid == 0 ) {           /* находимся в потомке */
        execlp("ls", "ls", (char*)0);
        exit(1); /* оказываемся здесь только в случае неудачного
                    исполнения вызова execlp */
    }
    else if ( child_pid != -1 ) {     /* находимся в предке */
        status = wait(&child_status);
        printf(" %d %d \n", status, child_status);
    }
}
```

В листинге 7.2 приведена простейшая программа на языке C, порождающая новый процесс в ОС Windows.

Листинг 7.2. Порождение процесса в ОС Windows

```
void main() {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    if ( !CreateProcess(NULL, "c:\\windows\\notepad.exe",
                        NULL, NULL, FALSE, 0, NULL, NULL,
                        &si, &pi) ) {
        ErrorExit("Не удалось создать процесс.");
    }
    WaitForSingleObject(pi.hProcess, INFINITE); // ждем завершения
    CloseHandle(pi.hProcess);                   // освобождаем ресурсы
    CloseHandle(pi.hThread);
}
```

7.2.1.2. Процессы с поддержкой многопоточности

Поток (нить) управления (thread) — это исполнение команд программы в естественном порядке. Процессы делятся на традиционные (имеющие один поток управления) и многопоточные (многонитевые).

Многопоточность в рамках одного процесса имеет существенные преимущества. Переключение контекста между двумя потоками в одном процессе

значительно проще, чем переключение контекста между двумя процессами. Все данные, за исключением стека исполнения и содержимого регистров процессора, разделяются между потоками. Таким образом, многопоточность предоставляет эффективный параллелизм.

Можно указать целые классы программ, где необходима многопоточность:

- операционные системы;
- сетевые серверы;
- встроенные системы;
- вычислительные программы.

Потоки могут быть реализованы как в пространстве пользователя, так и в пространстве ядра. В первом случае ядро ничего не знает о потоках и управляет обычными однопоточными процессами. При этом процессы могут иметь собственный алгоритм планирования потоков. Во втором случае потоки создаются и завершаются по системным запросам ядра.

Одна из наиболее элегантных смешанных реализаций потоков выполнена в ОС Solaris. Все потоки (нити) можно разделить на три класса (рис. 7.7).

- *Потоки ядра* являются базовыми потоками. Они располагаются в адресном пространстве ядра и непосредственно связаны с процессами.
- *Облегченные (легковесные) потоки* служат для организации пользователем нескольких потоков управления в адресном пространстве. Каждому облегченному потоку соответствует свой отдельный ядерный поток. Каждый облегченный поток может отдельно планироваться (на каждый процессор по потоку). Облегченные потоки следует рассматривать как диспетчеризуемые сущности.
- *Пользовательские потоки*, для создания которых пользователь работает со стандартной библиотекой. Пользовательские потоки связываются с облегченными потоками.

Об использовании потоков

Как правило, технику явного распараллеливания не следует использовать при прикладном программировании. Нужно переложить заботу о распараллеливании на компиляторы, возможно помогая ему директивами (например, OpenMP). Однако системному программисту знание и умение применять нити в своих программах необходимо.

Многие операционные системы, ориентированные на рабочие станции и персональные компьютеры, начиная с середины 80-х годов XX века, включают поддержку многопоточности. Впервые стандарт на потоки появился в 1995 году. Это был стандарт IEEE POSIX 1003.1c-1995. Однако стандарт появился достаточно поздно, и некоторые компании успели выпустить свои

версии многопоточных библиотек, существенно отличающихся от стандарта. Можно выделить следующие основные семейства потоков:

- потоки, поддерживающие стиль POSIX-стандарта. Это семейство состоит из трех подгрупп:
  - "истинные" POSIX-потоки, базирующиеся на стандарте IEEE POSIX 1003.1c-1995 (также известного как ISO/IEC 9945-1:1996), являющегося частью стандарта ANSI/IEEE 1003.1;
  - DCE-потоки, базирующиеся на ранней версии стандарта POSIX — 1003.1a;
  - Unix International потоки, также известные как Solaris-потоки [Lewis, Berg 1995]. Они достаточно близки к стандарту и поддерживаются в операционных системах Solaris компании Sun Microsystems, Inc и UnixWare 2 компании SCO;

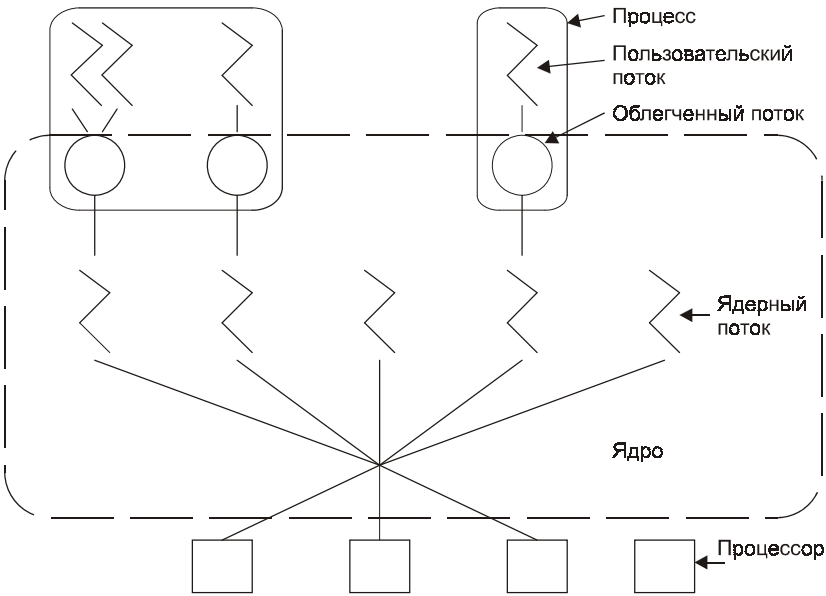


Рис. 7.7. Классы потоков

- потоки Microsoft. Это семейство состоит из двух подгрупп, причем обе разработаны в компании Microsoft:
  - потоки WIN32, являющиеся стандартными для семейства операционных систем Windows;
  - потоки OS/2, являющиеся стандартными для операционной системы OS/2 компании IBM.

Других вариантов потоков не так много. Отдельного упоминания заслуживает лишь пакет C threads, имеющийся в операционной системе Mach.

Различные семейства предлагают различный синтаксис основных функций потоковых библиотек. Основными группами функций являются функции создания потоков и функции синхронизации потоков (как правило, набор таких функций достаточно богат и разнообразен).

В листинге 7.3 приведена простейшая программа, демонстрирующая различие в синтаксисе функций потоков POSIX и Solaris. Компилировать ее надо с указанием нитевых библиотек, например:

```
% cc thread_example.c -lthread -lpthread
```

Листинг 7.3. Пример создания и использование потоков POSIX и Solaris

```
#define _REENTRANT
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);       /* функция, которая будет исполняться
                                 как поток */
int i;
thread_t tid[NUM_THREADS];    /* массив для хранения номеров потоков */

void main(int argc, char *argv[]) {
    switch ( *argv[1] ) {
        case '0':  /* POSIX */
            for ( i = 0; i < NUM_THREADS; i++ )
                pthread_create(&tid[i], NULL, sleeping, SLEEP_TIME);
            for ( i = 0; i < NUM_THREADS; i++ )
                pthread_join(tid[i], NULL);
            break;
        case '1':  /* Solaris */
            for ( i = 0; i < NUM_THREADS; i++ )
                thr_create(NULL,0,sleeping,NULL,0,&tid[i]);
            while (thr_join(NULL, NULL, NULL) == 0 )
                ;
            break;
    }  /* switch */
    printf("Все %d потоков управления завершены\n", i);
}
```

```
void *sleeping(int sleep_time *) {
    printf("Поток управления %d приостановлен на %d секунд\n",
           thr_self(), SLEEP_TIME);
    sleep(sleep_time);
    printf("Исполнение потока управления %d возобновлено\n", thr_self());
}
```

О безопасности использования фрагментов кода в многопоточных программах

В документации к некоторым фрагментам кода (как правило, к системным функциям) стоит пометка о том, что они безопасны с точки зрения использования в многопоточных программах (MT-safe). Отсутствие подобной пометки может означать наличие в таких функциях глобальных или статических переменных, приводящих к побочным эффектам.

7.2.1.3. Сигналы как простейшие средства коммуникации

Сигналы (signals) — это программное средство, с помощью которого может быть прервано функционирование процесса в операционной системе Unix. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире [Lewine 1995].

Каждому сигналу ставится в соответствие:

- номер сигнала;
- символическая константа, используемая для осмысленной идентификации сигнала;
- принадлежность к одному из пяти классов сигналов;
- реакция процесса на сигнал:
  - вызов обработчика сигнала;
  - завершение процесса (возможно, с порождением дампа памяти);
  - игнорирование сигнала;
  - приостановка исполнения.

Следующие три ситуации приводят к генерации сигналов операционной системой:

- ядро отправляет процессу сигнал при нажатии пользователем определенной комбинации клавиш на клавиатуре;
- возникновение аппаратных особых ситуаций приводит к посылке уведомления ядру операционной системы, которое отправляет сигнал процессу, находящемуся в стадии выполнения;
- определенное программное состояние системы может вызвать отправку сигнала.

Наиболее интересными являются следующие типы сигналов:

- *сигналы от оборудования*, порождаемые при выполнении процесса в результате возникновения некоторых исключительных условий. Вот некоторые примеры сигналов:
  - SIGFPE (8) — исключительная ситуация при выполнении операции с плавающей или фиксированной точкой;
  - SIGILL (4) — попытка выполнить неверную машинную команду;
  - SIGSEGV (11) — нарушение защиты памяти;
  - SIGBUS (10) — ошибка шины (адресации);
- *программные сигналы*, либо генерируемые пользователем с клавиатуры, либо посылаемые одним процессом другому. Примеры сигналов следующие:
  - SIGINT (2) — сигнал прерывания от терминала (пользователь нажал комбинацию клавиш <Ctrl> + <C>);
  - SIGQUIT (3) — сигнал аварийного завершения работы от терминала (пользователь нажал комбинацию клавиш <Ctrl> + <\>);
  - SIGTERM (15) — сигнал программного прерывания, посылаемого по умолчанию командой `kill`;
  - SIGKILL (9) — сигнал программного прерывания, который не может быть ни перехвачен, ни проигнорирован никаким процессом;
  - SIGHUP (1) — разрыв связи с терминалом;
- *сигналы контроля процессов*, либо генерируемые пользователем с клавиатуры, либо посылаемые одним процессом другому. Ряд этих сигналов используется отладчиками для посылки сигналов процессу с отлаживаемой программой. Примеры сигналов таковы:
  - SIGSTOP (23) — остановка процесса;
  - SIGTSTP (24) — остановка процесса пользователем (пользователь нажал комбинацию клавиш <Ctrl> + <z>);
  - SIGCONT (25) — продолжение приостановленного процесса;
  - SIGCHLD (18) — изменение статуса порожденного процесса.

В листинге 7.4 приведена программа на языке Pascal, исполнение которой приведет к генерации сигнала SIGSEGV.

Листинг 7.4. Пример порождения сигнала SIGSEGV

```
program SegmentationViolation;
type
    Pinteger = ^integer;
var
    IntVar:  integer;
    NullPtr: Pinteger;
```

```
begin
    NullPtr := nil;
  { Попытка выполнить следующее присваивание приведет к сигналу SIGSEGV }
    IntVar := NullPtr^;
end.
z
```

Листинг 7.5 содержит пример программы, устанавливающей обработчик сигнала прерывания от терминала.

Листинг 7.5. Пример перехвата и обработки сигнала прерывания от терминала

```
#include <signal.h>

static void reaction(int i) {
    printf("Вызван обработчик сигнала %d \n", i);
    exit(i);
}

void main() {
    if ( signal(SIGINT, SIG_IGN) != SIG_IGN )
        signal(SIGINT, reaction);
    printf("Перед подачей сигнала с клавиатуры \n");
    while ( 1 )
        ;
    printf("После подачи сигнала с клавиатуры \n");
}
```

Запуск на исполнение данной программы и последующее нажатие пользователем комбинации клавиш <Ctrl> + <C> приведет к выводу следующего результата работы программы:

```
% a.out
Перед подачей сигнала с клавиатуры
Вызван обработчик сигнала 2
```

Листинг 7.6 содержит более сложный пример программы, в котором после обработки сигнала прерывания от терминала происходит возвращение к работе программы. Для этого используется нелокальный переход.

Листинг 7.6. Пример использования стека восстановления

```
#include <signal.h>
#include <setjmp.h>
```

```
static jmp_buf jb;
static void reaction (int i) {
    printf("Вызван обработчик сигнала %d \n", i);
    longjmp(jb, i);
}

void main() {
    if ( signal(SIGINT, SIG_IGN) != SIG_IGN )
        signal(SIGINT, reaction);
    if ( !setjmp(jb) )
        printf("Ветка исполнения после setjmp \n");
    else
        printf("Ветка исполнения после longjmp \n");
    printf("Перед подачей сигнала с клавиатуры \n");
    while ( 1 )
        ;
}
```

Запуск на исполнение данной программы и последующее нажатие пользователем комбинации клавиш <Ctrl> + <C> приведет к выводу следующего результата работы программы:

```
% a.out
Ветка исполнения после setjmp
Перед подачей сигнала с клавиатуры
Вызван обработчик сигнала 2
Ветка исполнения после longjmp
Перед подачей сигнала с клавиатуры
%
```

7.2.2. Коммуникация и синхронизация процессов в централизованных архитектурах

7.2.2.1. Основные понятия и определения

Процессы называются *параллельными*, если они существуют одновременно. Параллельные процессы можно разделить на следующие две группы:

- *независимые* (не нуждающиеся во взаимодействии друг с другом) процессы;
- *асинхронные* (взаимодействующие и нуждающиеся в периодической синхронизации) процессы.

Синхронизация процессов — использование специальных атомических операций для осуществления взаимодействия между процессами.

Критический ресурс — это ресурс, допускающий обслуживание только одного процесса за один раз. Если несколько процессов хотят использовать критический ресурс в режиме разделения, то им следует синхронизировать свои действия, чтобы ресурс всегда находился в распоряжении не более чем одного из них.

Критические участки — это участки процесса, где происходит обращение к критическому ресурсу. Критические участки должны быть *взаимоисключаемыми*, т. е. в каждый момент времени не более чем один процесс может быть занят выполнением своего критического относительно некоторого ресурса участка. Обеспечение (поддержка механизма) взаимоисключения — ключевая задача параллельного программирования.

Блокировка — предотвращение выполнения кем-либо чего-либо. Процесс должен устанавливать блокировку перед входом в критический участок и снимать ее после выхода. Естественно, если участок заблокирован, то другой процесс должен ждать снятия блокировки.

Вход взаимоисключения и *выход взаимоисключения* — участки процесса, обрамляющие критический участок и служащие для обеспечения взаимоисключения. Примером входа взаимоисключения может служить блокирование, а выхода — разблокирование.

Типичным примером критического ресурса является разделяемая переменная, суммирующая некоторую величину (назовем ее `счетчик`). Критические участки процессов тогда могут содержать следующий код:

```
счетчик := счетчик + 1.
```

Гонки (race conditions) — ситуация, когда два или более процессов обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей их исполнения.

> ***Об использовании термина "процесс"***
>
> Мы будем использовать в определениях и алгоритмах этого раздела термин "процесс", следуя традиции. Более корректно было бы использовать понятие "поток управления".

7.2.2.2. Алгоритм Деккера

В листинге 7.7 приведена реализация *алгоритма Деккера* на языке C. Это программный вариант решения проблемы взаимного исключения. Алгоритм учитывает следующие требования:

- относительные скорости параллельных процессов могут быть любыми;
- процессы вне критического участка не могут препятствовать другим процессам входить в критический участок;
- не должно быть бесконечного откладывания входа в критический участок.

Листинг 7.7. Алгоритм Деккера

```
enum state { UNLOCKED, LOCKED };
typedef struct {
    char status[2];   /* байт статуса для каждого из двух процессов */
    char turn;        /* какой из процессов будет следующим */
} lock_t;

void init_lock(lock_t* lock) {
    lock->status[0] = UNLOCKED;
    lock->status[1] = UNLOCKED;
    lock->turn = 0;
}

void lock(volatile lock_t* lock) {
    /* устанавливаем блокировку для текущего процесса */
    lock->status[cur_proc_id()] = LOCKED;
    /* проверяем, не установлена ли блокировка другим процессом */
    while ( lock->status[other_proc_id()] == LOCKED ) {
       /* если другой процесс уже установил блокировку,
          проверяем — чья очередь войти в критический участок */
       if ( lock->turn != cur_proc_id() ) {
           lock->status[cur_proc_id()] = UNLOCKED;
           while ( lock->turn == other_proc_id() )
               ;
           lock->status[cur_proc_id()] = LOCKED;
       }
    }
}

void unlock(lock_t* lock) {
    lock->status[cur_proc_id()] = UNLOCKED;
    lock->turn = other_proc_id();
}
```

7.2.2.3. Аппаратная поддержка взаимоисключений

Возможное *аппаратное решение поддержки проблемы взаимного исключения* — это наличие неделимой команды `test_and_set(a, b)` (проверить и установить).

Команда имеет две логические переменные в качестве параметров. Переменная `a` является локальной для каждого процесса, а переменная `b` — глобальная и разделяется процессами. Выполнение команды заключается в следующих двух действиях:

- значение переменной `b` копируется в `a`;
- значение переменной `b` устанавливается в истину.

Практически все основные архитектуры имеют подобную команду в своем составе. Например, в SPARC-архитектуре существует команда `ldstub` (load store unsigned byte) *(см. разд. 6.4.3.6)*:

```
ldstub [addr],reg
```

В результате выполнения этой команды содержимое памяти по адресу `addr` копируется в регистр `reg`, а все биты памяти `addr` устанавливаются в единицу. В листинге 7.8 представлен программный интерфейс команды `test_and_set` с помощью реальной ассемблерной команды `ldstub`.

Листинг 7.8. Программный интерфейс команды `test_and_set`

```
int test_and_set(volatile int* addr) {
    asm(ldstub [addr],reg);
    if ( reg == 0 )
        return 0;
    return 1;
}
```

7.2.2.4. Крутящаяся блокировка

Крутящаяся блокировка — это механизм, реализующий взаимное исключение и являющийся достаточно эффективным для коротких критических участков. Его название вытекает из того факта, что процесс будет находиться в цикле, ожидая завершение блокировки, установленной другим процессом. В листинге 7.9 программная реализация крутящейся блокировки использует неделимую команду `test_and_set`.

Листинг 7.9. Крутящаяся блокировка

```
typedef int lock_t;

void init_lock(lock_t* lock_status) {
    *lock_status = 0;
}
```

```
void lock(volatile lock_t* lock_status) {
    while ( test_and_set(lock_status ) == 1 )
        ;
}
void unlock(lock_t* lock_status) {
    *lock_status = 0;
}
```

7.2.2.5. Блокировка с запретом прерываний

Блокировка с запретом прерываний применима только на однопроцессорной архитектуре. В листинге 7.10 приведен пример реализации такой блокировки.

Листинг 7.10. Блокировка с запретом прерываний

```
enum state { UNLOCKED, LOCKED };
typedef int lock_t;

void init_lock(lock_t* lock) {
    *lock = UNLOCKED;
}
void lock(volatile lock_t* lock) {
    disable_interrupts; /* запрет прерываний */
    while ( lock != UNLOCKED ) {
        enable_interrupts; /* разрешение прерываний */
        disable_interrupts;
    }
    lock = LOCKED;
    enable_interrupts; /* разрешение прерываний */
}
void unlock(lock_t* lock) {
    disable_interrupts;
    lock = UNLOCKED;
    enable_interrupts;
}
```

7.2.2.6. Семафоры

Семафор — это защищенная переменная, значение которой можно запрашивать и менять только при помощи специальных операций `P` и `V` и при

инициализации. Концепция семафоров была предложена Дейкстрой в начале 60-х годов XX века. Применяют три основных типа семафоров:

- *двоичные* (бинарные) семафоры, принимающие только два значения {0, 1};
- *считающие* семафоры. Их значения — целые неотрицательные числа;
- *общие* семафоры. Принимают все множество целых чисел.

Реализация операций `P` и `V` для считающих семафоров выглядит так:

```
P(S):  if ( S>0 )
       then S:=S-1
       else ожидать_в_очереди(S)

V(S):  if ( есть_процессы_в_очереди(S) )
       then одному_продолжить(S)
       else S:=S+1.
```

Конечно, операции `P` и `V` должны быть неделимыми (например, защищенные крутящейся блокировкой). Как правило, семафоры реализуются в ядре операционной системы. Собственно название операций происходит от голландских слов Proberen — проверить и Verhogen — увеличить.

Три классические задачи, в которых применяются семафоры, таковы:

- решение проблемы взаимного исключения при помощи семафоров. Входом взаимоисключения будет операция `P`, а выходом — `V`;
- синхронизация при помощи семафоров. Если одному процессу необходимо, чтобы он получал уведомление от другого процесса о наступлении некоторого события и только после этого продолжал свою работу, то процессы должны иметь операции `P` и `V` соответственно, причем инициализироваться семафор должен нулем;
- выделение нескольких однотипных ресурсов из пула ресурсов при помощи семафоров. В этом случае надо применять общие или считающие семафоры.

Одна из возможных реализаций семафоров представлена в листинге 7.11.

Листинг 7.11. Реализация семафоров

```
typedef struct {
    lock_t    lock;   /* байт статуса для каждого из двух процессов */
    int       count;  /* какой из процессов будет следующим */
    proc_t*   head;
    proc_t*   tail
} sema_t;
```

```
void init_sema(sema_t* sema, int initial_count) {
    init_lock(&sema->lock);
    sema->head = NULL;
    sema->tail = NULL;
    sema-> count = initial_count;
}
void P(sema_t* sema) {
    lock(&sema->lock);
    sema->count--;
    if ( sema->count < 0 ) {
        if ( sema->head == NULL )
            sema->head = u.u_procp;
        else
            sema->tail->p_next = u.u_procp;
        u.u_procp->p_next = NULL;
        sema->tail = u.u_procp;
        unlock(&sema->lock);
        switch();
        return;
    }
    unlock(&sema->lock);
}
void V(sema_t* sema) {
    proc_t* p;
    lock(&sema->lock);
    sema->count++;
    if ( sema->count <= 0 ) {
        p = sema->head;
        sema->head = p->p_next;
        if ( sema->head == NULL )
            sema->tail = NULL;
        unlock(&sema->lock);
        enqueue(&runqueue, p);
        return;
    }
    unlock(&sema->lock);
}
```

7.2.2.7. Мониторы

Монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для динамического распределения конкретного общего ресурса или группы общих ресурсов.

Для выделения нужного ресурса процесс должен обратиться к конкретной процедуре монитора. В каждый конкретный момент времени монитором может пользоваться только один процесс. Процессам, желающим войти в монитор в тот момент, когда он занят, приходится ждать, причем процессом ожидания управляет монитор. Монитор обладает свойством инкапсуляции данных.

Принцип работы монитора может быть описан следующим образом.

- Процесс, обращающийся к монитору за получением некоторого ресурса, обнаруживает, что ресурс занят. При этом процедура монитора выдает команду "ждать" (wait), по которой процесс будет ждать вне монитора, пока ресурс освободится.
- Когда ресурс освобождается, то монитор выдаст команду "сигнал" (signal). Если очередь ждущих процессов не пуста, то по этой команде один из процессов может воспользоваться ресурсом монитора. Обычно очередь организуется по принципу "первый пришел — первый получил доступ к ресурсу".

Существует традиционный подход к реализации мониторов и очереди процессов. Очередь поддерживается переменной условия (condition variable). При этом в команды включаются имена условий, например, `wait(имя_условия)` или `signal(имя_условия)`.

В листинге 7.12 приведена реализация двоичного семафора при помощи монитора. Пример написан на языке, близком к языку Pascal, но расширенному конструкцией `monitor`.

Листинг 7.12. Реализация двоичного семафора при помощи монитора

```
monitor sema;
var
    s: integer = 1; /* переменная, представляющая семафор */
    res_is_free: cond_t;
procedure P;        /* захват ресурса */
begin
    if s = 1
    then s := 0
    else wait(res_is_free)
end;
```

```
procedure V;        /* освобождение ресурса */
begin
    s := 1;
    signal(res_is_free)
end;
end monitor;
```

7.2.2.8. Рандеву в языке программирования Ada

В языке программирования Ada существует специальный встроенный механизм параллельного программирования — `task` (задача). Одни из задач могут обращаться к другим с помощью механизма вызовов и обслуживания входов. *Рандеву* происходит в момент, когда вызов входов принимается оператором `accept` и представляет собой выполнение операторов. Точка входа может обслужить только одну связанную с ней точку вызова. Остальные будут помещены в очередь ожидания.

7.2.2.9. Решение задачи передачи данных между процессами "читатель-писатель"

Существует классическая задача передачи данных между процессами "читатель-писатель". Процесс "писатель" генерирует информацию, а "читатель" ее потребляет. Обмен блоками информации осуществляется через специальные буферы, число которых ограничено. Одним из возможных вариантов буфера может быть циклический список. В листингах 7.13—7.16 эта задача решается с помощью блокировок, семафоров, мониторов и средств языка программирования Ada, соответственно.

Листинг 7.13. Решение задачи с помощью блокировок

```
char buf[N];
int head = 0, tail = 0, n = 0;
lock_t lock_status;

void put(char c) {
    while ( n == N )
        wait();
    lock(lock_status);
    buf[head] = c;
    head = (head + 1)%N;
    n++;
    unlock(lock_status);
}
```

```
char get() {
    while ( n == 0 )
        wait();
    lock(lock_status);
    c = buf[tail];
    tail = (tail + 1)%N;
    n--;
    unlock(lock_status);
    return c;
}
```

Листинг 7.14. Решение задачи с помощью семафоров

```
char buf[N];
int head = 0, tail = 0;
sema_t holes, chars;

init_sema(holes, N);
init_sema(chars, 0);
void put(char c) {
    P(holes);
    buf[head] = c;
    head = (head + 1)%N;
    V(chars);
}
char get() {
    P(chars);
    c = buf[tail];
    tail = (tail + 1)%N;
    V(holes);
    return c;
}
```

Листинг 7.15. Решение задачи с помощью мониторов

```
char buf[N];
int head = 0, tail = 0, n = 0;
cond_t not_empty, not_full;
```

```
void put(char c) {
    while ( n == N )
        wait(not_full);
    buf[head] = c;
    head = (head + 1)%N;
    n++;
    signal(not_empty);
}
char get() {
    while ( n == 0 )
        wait(not_empty);
    c = buf[tail];
    tail = (tail + 1)%N;
    n--;
    signal(not_full);
    return c;
}
```

Листинг 7.16. Решение задачи на языке программирования Ada

```
task body Buffer is
    buf: array(0..N-1) of CHAR;
    head, tail: INTEGER range 0..N-1 := 0;
    n: INTEGER range 0..N := 0;
begin
    loop
       select
           when n < N => accept put(in c: CHAR) do
              buf[head] = c;
           end;
           n +:= 1;
           head = (head + 1) mod N;
       or
           when n > 0 => accept get(out c: CHAR) do
              c = buf[tail];
           end;
           n -:= 1;
           tail = (tail + 1) mod N;
```

```
        end select;
    end loop;
end Buffer;
```

Существует частный случай задачи передачи данных между процессами "читатель-писатель". Это случай информационной базы данных, который характеризуется следующим образом. Процессов-"читатель" гораздо больше, чем процессов-"писатель". "Читатели" не изменяют содержимое базы данных, следовательно, их может быть несколько, при этом они одновременно имеют доступ к базе данных. "Писатели" должны иметь монопольный доступ к базе данных.

7.2.2.10. Тупики

Процессы и потоки управления — активные объекты. Ресурсы — неактивные объекты (процессор — вытесняемый ресурс, а дисковое пространство — невытесняемый ресурс). Во время своей работы процесс (поток управления) может попасть в два неприятных состояния: зависание и тупик.

Зависание — это состояние неопределенного ожидания каких-либо ресурсов, из которого рано или поздно процессы выходят.

Тупик — это состояние ожидания некоторого события, которое никогда не произойдет (как правило, это круговое ожидание ресурсов).

Система находится в тупиковой ситуации, если один или более процессов находятся в состоянии тупика.

Существуют четыре необходимых условия для возникновения тупика.

- Условие *взаимоисключения*. Процессы требуют монопольного владения ресурсами, предоставляемыми им.
- Условие *ожидания*. Процессы удерживают уже выделенные им ресурсы, ожидая выделения дополнительных.
- Условие *нераспределяемости*. Ресурсы нельзя отобрать у удерживающих их процессов, пока они не будут использованы.
- Условие *кругового ожидания*. Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу.

Существует четыре основных стратегии работы с тупиками.

- *Полное игнорирование проблемы* ("алгоритм страуса") — подход, при котором операционные системы не осуществляют борьбу с тупиками. Большинство реальных систем поступает именно так, поскольку ресурсов достаточно много.

- *Предотвращение тупиков* (prevention) — подход, целью которого является обеспечение условий, исключающих возможность возникновения тупиковой ситуации. Чтобы предотвратить тупик, достаточно нарушить хотя бы одно необходимое условие.
  - Первое условие (взаимоисключение) вполне естественно (например, для такого устройства, как накопитель на магнитной ленте).
  - Нарушая второе условие (ожидание), процесс должен сразу запросить все ресурсы. Эффективность системы при этом может значительно ухудшиться. В качестве компромиссного решения можно предложить разделять процесс на шаги.
  - Нарушить третье условие (нераспределяемость) можно следующим образом. Процесс, удерживающий ресурсы и получивший отказ на другие ресурсы, должен освободить все взятые ресурсы и через некоторое время запросить их заново. При этом процесс может потерять большую часть проделанной работы.
  - Нарушая четвертое условие (кругового ожидания), следует ввести линейную упорядоченность ресурсов, пронумеровав их, и выдавать их в порядке, не допускающем возникновения кругового ожидания. Данный подход требует значительных накладных расходов на хранение информации, связанной с типами ресурсов и упорядоченностью экземпляров ресурсов.
- *Обход тупиков* (avoidance) — подход, который обеспечивает рациональное распределение ресурсов по рациональным правилам. Он вводит менее жесткие ограничения, чем предыдущий подход. Наиболее известным методом обхода тупиков является алгоритм банкира. Перечислим основные идеи, лежащие в его основе.
  - Алгоритм банкира имитирует действия банкира, располагающего определенным источником капитала, выдающего ссуды и принимающего платежи. Алгоритм был предложен Дейкстрой.
  - Состояние системы будем называть надежным, если операционная система может обеспечить всем пользователям завершение их задач в течение конечного промежутка времени. Суть алгоритма в том, что система удовлетворяет только те запросы, при которых ее состояние остается надежным. Остальные запросы откладываются.
  - Существуют два основных ограничения этого подхода: каждый процесс заранее должен указать максимальное количество ресурсов, которое ему понадобится, и в каждый момент времени процесс должен захватывать только один ресурс. Данный алгоритм на практике неприменим из-за его неэффективности, т. к. необходимость в пересчете возникает непрерывно, при каждом поступлении процесса в систему и его удалении для каждой разновидности ресурсов.

- *Обнаружение тупиков* (detection) — подход, который допускает возникновение тупиков, определяет процессы и ресурсы, которые вовлечены в тупиковую ситуацию, и пытается вывести систему из нее.

7.2.2.11. Модели для анализа свойств асинхронных процессов

Для анализа свойств асинхронных процессов (в частности, для обнаружения тупиков) используются графические модели, которые мы сейчас рассмотрим.

Граф распределения ресурсов

Задачу установления факта возникновения тупиковой ситуации удобно решать с помощью графа распределения ресурсов. На рис. 7.8 представлены два основных типа отношений (рис. 7.8, *а*) на графе (запрос ресурса и владение им) и простейший пример системы в состоянии кругового ожидания (тупика) (рис. 7.8, *б*).

а *б*

Рис. 7.8. Примеры графа распределения ресурсов: типы отношений (*а*), состояние тупика (*б*)

Для обнаружения тупиков выполняется редукция (приведение) графа. Существует правило редукции: для каждого незаблокированного процесса (т. е. процесса, все запросы которого могут быть удовлетворены) нужно убрать все входящие и исходящие дуги. Граф полностью приводим, если после редукции он не содержит ни одной дуги. В системе отсутствуют тупиковые ситуации, если соответствующий граф полностью приводим.

Сеть Петри

Сеть Петри — это помеченный ориентированный граф с двумя типами вершин: позициями и переходами. Позиции изображаются кругами, переходы — квадратами, а пометки — жирными точками в позициях.

Разметка сети Петри — функция, которая ставит в соответствие пометкам позиции целое неотрицательное число. Разметка может быть изменена с помощью срабатывания (запуска) перехода. Переход называется *запускаемым*, если в каждой позиции, из которой ведет стрелка в данный переход, есть хотя бы одна метка. Запуск перехода заключается в том, что из каждой позиции, из которой ведет стрелка, число пометок уменьшается на

единицу. А в каждую позицию, в которую ведет стрелка, число пометок увеличивается на единицу.

Семантически позиции удобно рассматривать как некоторые условия, а переходы — как некоторые события, происходящие в системе. Разметка называется *живучей*, если каждый из переходов в системе может быть запущен бесконечное число раз. Когда достигается разметка, при которой ни один из переходов не может быть запущен, говорят, что сеть Петри "зависла". Если разметка живучая, то система не остановится.

На рис. 7.9 выполнено моделирование взаимного исключения между двумя процессами с помощью сети Петри.

Позиции, помеченные `КУ1` и `КУ2`, представляют соответственно критические участки первого и второго процесса. При текущей разметке могут быть запущены переходы `P11` или `P21`. Если запускается переход `P11`, то позиция `КУ1` получает пометку (система входит в критический участок). При этом второй процесс не может войти в `КУ2`, поскольку разделяемая позиция является пустой. Теперь может быть запущен только переход `P12`. После его запуска система возвращается в исходное состояние.

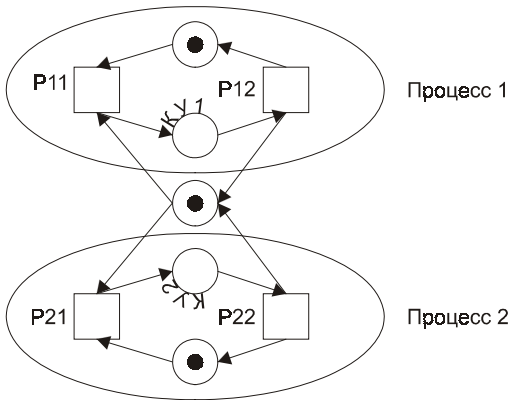


Рис. 7.9. Моделирование взаимного исключения с помощью сети Петри

Вычислительные схемы

Сеть Петри не обеспечивает полной общности с вычислительной точки зрения. Например, сетью Петри нельзя смоделировать оператор отрицания НЕ. Существует еще одна структура под названием "вычислительная схема", которая эту проблему решает.

Вычислительная схема — это представление в графической форме асинхронной системы, состоящей из набора операторов (процессов), воздействующих на множество регистров (ячеек памяти). Вычислительная схема поддерживает две структуры: граф потока данных и граф потока управления.

7.2.2.12. Планирование и диспетчеризация процессов

С точки зрения распределения процессорного времени операционные системы можно разделить на три группы.

- *Системы с однопользовательским режимом*. Система запускает одну задачу и ждет ее полного завершения.
- *Системы с пакетным режимом*. Переключение процессора с одной задачи на другую происходит лишь в том случае, если активная задача сама отказывается от процессора.
- *Системы с многозадачным режимом*. Переключение процессора происходит по истечении некоторого кванта времени. Многозадачный режим применяется в интерактивных системах и системах реального времени.

Выделяют три уровня планирования.

- *Планирование верхнего уровня* — это планирование при поступлении в систему, планирование стадии "оформления" процесса и его допуска в систему.
- *Планирование промежуточного уровня* — это планирование при переводе процесса из очереди ожидающих ресурсы в очередь готовых к помещению на процессор.
- *Планирование нижнего уровня (диспетчеризация)* — это планирование очереди готовых к помещению на процессор процессов.

Можно сформулировать пять основных целей планирования.

- Справедливость планирования, заключающаяся в том, что надо относиться к процессам одинаково и не откладывать бесконечно их поступление на процессор.
- Завершение максимального количества процессов в единицу времени. Эта цель особенно актуальна для пакетных систем.
- Обеспечение приемлемого времени ответа максимальному числу пользователей. Цель особенно актуальна для интерактивных систем.
- Предсказуемость планирования, определяемая тем, что одна и та же задача должна выполняться в системе за одно и то же время, независимо от условий. Цель особенно актуальна для систем реального времени.
- Постепенное снижение работоспособности системы.

Приоритеты

Приоритет — некоторое число, сопоставленное каждому процессу из очереди готовых процессов и обозначающее важность процесса. Приоритеты бывают:

- статические (не меняющиеся с момента поступления процесса в систему) и динамические;
- присваиваемые автоматически (system defined) и назначаемые извне (user defined);

- купленные (например, в 70-х годах XX века в американских вычислительных центрах за назначение высокого приоритета платили деньги) и заслуженные;
- рациональные (назначенные из разумных соображений) и случайные (random).

Алгоритмы планирования

Перечислим основные алгоритмы планирования, ориентированные на пакетные операционные системы.

- Первый, пришедший в очередь процесс, обслуживается первым (First Comes First Served — FCFS). Процессы получают процессор в порядке их поступления в очередь готовых процессов. Получив процессор, они выполняются на нем до своего полного завершения.
- Кратчайший процесс обслуживается первым (Shortest Job First — SJF). На процессор первым поступает процесс с минимальным оценочным временем исполнения. Процессы исполняются до полного завершения. Заметим, что в большинстве случаев невозможно предсказать оценочное время завершения. Возможным вариантом решения является сохранение истории и анализ косвенных признаков (число обращений к внешним устройствам).
- Первым обслуживается процесс с наименьшим остаточным временем (Shortest Rest Time — SRT). Это случай дополнения предыдущего алгоритма квантованием времени.

Два следующих алгоритма планирования ориентированы на интерактивные операционные системы.

- Циклическое (круговое) обслуживание (Round Robbin — RR). Каждый процесс находится на процессоре ограниченный квант времени, по истечении которого становится в конец очереди. Разновидность кругового обслуживания — круговорот со смещением, при котором квант времени зависит от внешнего приоритета процесса.
- Многоуровневая очередь с обратными связями. Можно сказать, что данный алгоритм использует прошлое, чтобы предсказать будущее. Вначале каждый процесс попадает в очередь с одинаковым приоритетом. Если процесс провел весь положенный ему квант времени на процессоре, то он переходит в очередь с меньшим приоритетом. Если процесс не отработал весь квант времени, то он переходит в очередь с большим приоритетом. Высший приоритет получают те задачи, которым он, как правило, нужен (например, интерактивные). Сложные вычислительные задачи, занимающие много времени, попадают в очередь с небольшим приоритетом.

7.2.3. Коммуникация процессов в сетях

7.2.3.1. Уровневые протоколы

Протокол — это совокупность синтаксических и семантических правил, которые определяют поведение функциональных блоков при передаче данных.

Уровень — компонент иерархической структуры.

Назначение уровневых протоколов таково:

- обеспечение логической декомпозиции сложной сети на меньшие (и более понятные) части и уровни;
- обеспечение симметрии в отношении функций, которые реализуются в каждом узле сети;
- обеспечение стандартного интерфейса между сетевыми функциями.

Уровень является поставщиком сервиса и может состоять из нескольких сервисных функций. Каждый уровень преобразует полученную информацию. Верхний уровень обеспечен полным набором услуг, предлагаемых всеми нижними уровнями.

Семиуровневая модель протоколов Взаимосвязи Открытых Систем

Сначала рассмотрим *семиуровневую модель протоколов Взаимосвязи Открытых Систем* — ВОС (Open Systems Interconnections — OSI), разработанную институтом ISO. Семь уровней этой модели и их функции таковы:

1. *Физический* уровень. Отвечает за передачу неструктурированного потока данных по физической среде (например, коаксиальному кабелю, витой паре или оптоволокну).
2. *Канальный* уровень. Обеспечивает прием и передачу пакетов, определение аппаратных адресов.
3. *Сетевой* уровень. Выполняет маршрутизацию и ведение учета.
4. *Транспортный* уровень. Обеспечивает корректную сквозную передачу данных.
5. *Сеансовый* уровень. Осуществляет проверку полномочий (аутентификацию).
6. Уровень *представления*. Выполняет интерпретацию данных (например, определение кодировки).
7. *Пользовательский* уровень (уровень приложения). Предоставляет услуги конечному пользователю.

Существует реализация набора протоколов — GOSIP (Government OSI Profile), которой пользовались только правительственные организации США.

Уже после того, как эта модель была разработана, выяснилось, что [Немет, Снайдер, Сибасс, Хейн 1999]:

- эти протоколы разрабатывались на основе концепций, практически не имеющих смысла в современных сетях;
- их спецификации были в ряде случаев неполны;
- по своим функциональным возможностям они уступали реально существующим протоколам;
- из-за наличия большого количества уровней они были медлительны и трудны для реализации.

Как ни велика роль этой модели, считается, что данная семиуровневая модель стала одной из самых неудачных разработок комитетов по стандартизации.

Протоколы TCP/IP

В 1969 году появился стек протоколов TCP/IP (Transfer Control Protocol/Internet Protocol). Он содержит всего четыре уровня.

- Уровень *сетевого протокола* (эквивалентен 1 и 2-му уровням в модели BOC) — Ethernet, Token ring.
- Уровень *Интернета* (эквивалентен 3-му уровню в модели BOC) — IP.
- *Транспортный* уровень (эквивалентен 4-му уровню в модели BOC) — TCP, UDP.
- Уровень *приложений* (эквивалентен 5, 6 и 7-му уровням в модели BOC) — FTP, SMTP, TELNET, NFS (перечень файлов, электронная почта, терминал).

Приведем названия основных протоколов стека TCP/IP (рис. 7.10). Подробно все эти протоколы замечательно описаны в книге [Stevens 1994].

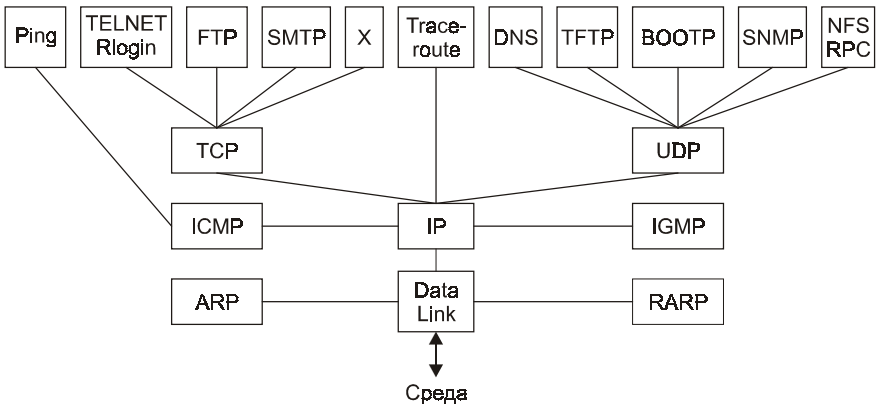


Рис. 7.10. Основные протоколы стека TCP/IP

Обмен данными между коммуникационными узлами

Она оседлала его и приготовилась выполнить необходимое соединение портов:
штыревая и розеточная части подготовлены, ввод-вывод разрешен,
режимы клиент-сервер, ведущий-ведомый.

Р. Дулинг. "Мозговой штурм"

Для того чтобы однозначно передать сообщение, отправитель должен знать:

- *способ соединения*. Для стека протоколов TCP/IP это протокол, определяемый транспортным уровнем;
- *адрес узла-адресата*. Для стека протоколов TCP/IP это IP-адрес;
- *номер порта*. Прикладной процесс, предоставляющий некоторые услуги другим прикладным процессам, ожидает поступления сообщений в порт, специально выделенный для этих услуг.

Данные пользователя, проходя по стеку протоколов при отправлении, дополняются заголовками на каждом уровне (рис. 7.11).

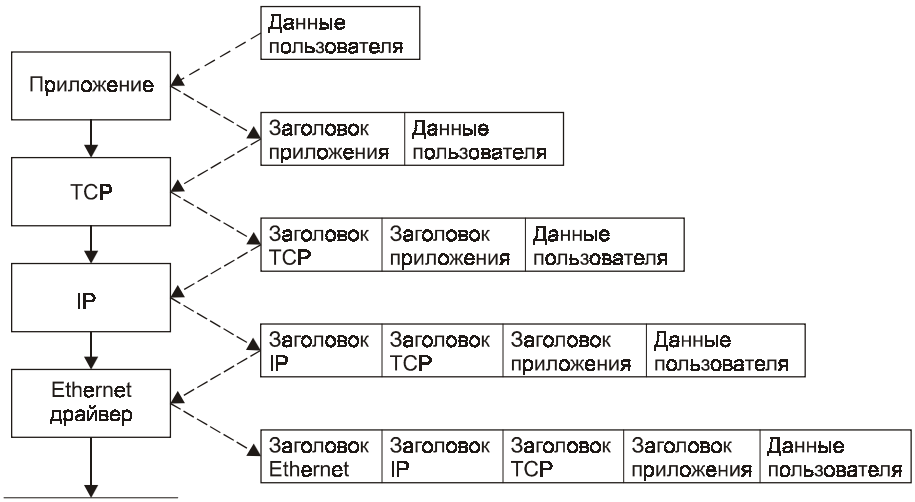


Рис. 7.11. Обмен данными в соответствии с протоколами стека TCP/IP

7.2.3.2. Адресация в сетях TCP/IP

Протокол TCP/IP использует следующую схему адресации [Немет, Снайдер, Сибасс, Хейн 1999].

- Нижний уровень адресации задается сетевыми аппаратными средствами. Сетевым картам Ethernet при их изготовлении задаются уникальные 6-байтные аппаратные адреса.
- Следующий уровень использует IP-адресацию. Каждому включенному в сеть узлу присваивается уникальный четырехбайтный IP-адрес. Этот

адрес глобален, уникален и не зависит от аппаратных средств. Основное назначение этих адресов заключается в содействии маршрутизации пакетов из одной физической сети в другую. Можно задать соответствие между IP-адресами и аппаратными адресами, реализуемое на канальном уровне модели TCP/IP. Преобразованием адресов занимаются протоколы ARP (Address Resolution Protocol) и RARP (Reverse Address Resolution Protocol).

- Верхний уровень адресации использует номера портов, служащие для определения процесса, которому адресованы данные и который выполняется на данном узле.

IP-адресация

IP-адрес — это уникальный адрес, идентифицирующий узлы или компьютеры в сети, управляемой протоколами TCP/IP.

Он состоит из следующих частей:

- *сетевой* части, обозначающей логическую сеть, к которой относится адрес. На основе этой части принимается решение о маршрутизации;
- *машинной* части, характеризующей конкретную машину в сети.

IP-адреса могут быть записаны как десятичные числа, разделенные точками. Существует несколько классов IP-адресов. Их отличие заключается в распределении байтов между сетевой и машинной частью. Класс адреса можно определить по его первому байту:

- класс A: 1—126 (крупная сеть);
- класс B: 128—191 (большие сети с подсетями);
- класс C: 192—223 (сети не более чем из 254 компьютеров).

Значения первого байта 0, 127 и 255 являются специальными и не используются для обычных IP-адресов. Если первый байт находится в диапазоне 224—239, то это групповая адресация, а если в диапазоне 240—254, то это экспериментальные адреса.

О сетевой и машинной частях

Необходимость различать класс адреса (выделяя сетевую и машинную часть) требуется для того, чтобы была возможность осуществить маршрутизацию. Подробно вопросы маршрутизации, в том числе бесклассовой, будут рассмотрены в *разд. 7.2.3.4*.

Общее число сетевых адресов классов A, B и C едва превышает два миллиона. Существующий интернет-протокол IPv4 с его 32-разрядной схемой адресации не может предоставить достаточное число адресов для всех возможных участников сетевого обмена. Беспрецедентный рост Интернета приводит к необходимости перехода на новый протокол IPv6 [Алексеев, Захарова, Русаков 2000]. Новый протокол имеет 128-разрядные адреса, что теоретически позволяет адресовать 240 триллионов узлов.

Номера портов

Номера портов служат для определения процесса, который выполняется на данном узле и которому адресованы данные. В табл. 7.1 приведены некоторые зарезервированные номера портов для широко известных приложений.

Таблица 7.1. *Некоторые зарезервированные номера портов*

Номер	Имя и описание сервиса, использующего данный порт
7	ECHO — пересылка пакетов обратно отправителю
20	FTP (File Transfer Protocol (Default Data)) — передача данных по протоколу передачи файлов
21	FTP (File Transfer Protocol (Control)) — управляющие команды протокола передачи файлов
23	TELNET — удаленный доступ
25	SMTP (Simple Mail Transfer Protocol) — электронная почта
53	DNS (Domain Name Server) — сервер доменных имен
79	FINGER — возвращение информации об одном или нескольких пользователях на указанном компьютере
111	RPC (Remote Procedure Call) — удаленный вызов процедур
115	SFTP (Simple File Transfer Protocol) — простой протокол передачи файлов
123	NTP (Network Time Protocol) — протокол синхронизации времени
161	SNMP (Simple Network Management Protocol) — простой протокол управления сетью

7.2.3.3. Транспортные протоколы

В стеке протоколов TCP/IP существуют два основных транспортных протокола: TCP и UDP (User Datagram Protocol).

Протокол UDP

Протокол UDP обеспечивает пакетную передачу данных между источником и получателем без предварительного установления связи (т. е. сообщения, обрабатываемые протоколом, не имеют друг к другу никакого отношения с точки зрения UDP). Для доставки сообщений используется протокол IP. Надежность в протоколе UDP отсутствует, и должна быть выполнена ее поддержка на уровне приложений. Данный протокол требует меньше накладных расходов, чем TCP. Обычно UDP используют такие протоколы верхнего уровня, как DNS и NTP.

Протокол TCP

Протокол TCP поддерживает надежную передачу потока данных с предварительной установкой связи между источником информации и ее получателем. Он используется такими протоколами верхнего уровня, как TELNET и FTP.

Отличительные черты TCP таковы:

- перед фактической передачей данных выполняется установка связи, т. е. запрос на начало передачи данных источником и подтверждение о готовности их принять получателем;
- доставка информации является надежной и не допускающей дублирования или изменения очередности получения данных;
- возможна доставка экстренных данных.

За счет усложнения транспортного уровня можно упростить уровень приложения. TCP поддерживает двунаправленный поток данных. Управление потоком данных осуществляется с помощью метода "скользящего окна". Начальная фаза сеанса носит название "тройного рукопожатия": одна из сторон посылает пакет, вторая подтверждает прием, тогда первая сторона посылает еще один пакет.

7.2.3.4. Маршрутизация в сетях TCP/IP

Маршрутизация — это выбор маршрута следования информации от источника к получателю через объединенную коммуникационную сеть.

С точки зрения графов маршрутизацию можно рассматривать как ориентированный граф, нагруженный по дугам. *Маршрут* — это путь, по которому движется сообщение (набор дуг). *Оптимальный маршрут* — это маршрут, для которого сумма весов всех дуг, его составляющих, минимальна. Для решения задач нахождения оптимального маршрута созданы алгоритмы маршрутизации.

Данные маршрутизации хранятся в одной из таблиц ядра. Каждая строка этой таблицы содержит несколько параметров, например:

- IP-адрес сети назначения;
- IP-адрес следующего узла (обычно называемого шлюзом или маршрутизатором), через который нужно посылать пакеты, чтобы достигнуть сети назначения;
- стоимость пути до назначения.

Протокол маршрутизации используется демонами маршрутизации для обмена информацией о сети и создается на базе алгоритма маршрутизации. Задача протокола — следить за правильной работой алгоритма.

Метрика — применяемая система измерения весов связей, которая в реальной работе может выступать как длина маршрута, время задержки, пропускная способность.

Маршрутизатор — это устройство, единственной функцией которого является распределение сетевого трафика или потока сообщений. Получая пакет, маршрутизатор направляет его либо по назначению, либо обратно, если узел недостижим.

Классификация алгоритмов маршрутизации

Существуют различные варианты разделения алгоритмов маршрутизации по категориям.

Например, алгоритмы могут быть:

- внутреннего или внешнего шлюза:
  - алгоритмы *внутреннего шлюза (внутридоменные)* управляют данными маршрутизации в пределах логических групп узлов, называемых доменами, или автономными системами, или областями;
  - алгоритмы *внешнего шлюза (междоменные)* отвечают за маршрутизацию между автономными системами. Природа этих двух типов алгоритмов различная. Поэтому понятно, что оптимальный алгоритм для внутридоменной маршрутизации не обязательно будет оптимальным для междоменной;
- статическими или динамическими:
  - при применении *статических* (неадаптивных) алгоритмов выбор маршрутов осуществляется заранее и заносится вручную в таблицу маршрутизации, где хранится информация о том, в каком направлении, какому соседу отправить пакет, следующий к определенному узлу назначения. Протоколы, разработанные на базе статических алгоритмов, называют также немаршрутизируемыми протоколами;
  - в случае *динамических* алгоритмов таблица маршрутизации меняется автоматически при изменении топологии сети или трафика в ней. Динамические алгоритмы отличаются по способу получения информации о состоянии сети, времени изменения маршрутов и по используемым показателям оценки маршрута;
- одномаршрутными или многомаршрутными:
  - *одномаршрутные* алгоритмы определяют один маршрут движения информации к получателю, и не всегда этот маршрут оказывается оптимальным. Такие алгоритмы отличаются критерием оценки маршрута;
  - *многомаршрутные* алгоритмы вырабатывают множество маршрутов к одному и тому же получателю. Они делают возможной мультиплексную передачу трафика по многочисленным каналам связи, обеспечивая при этом значительно большую пропускную способность и надежность сети;

- одноуровневыми или иерархическими:
  - в *одноуровневой* сети нет различия между разными ее частями. Все сегменты сети находятся на одном логическом уровне. *Иерархическая* сеть, как правило, состоит из двух частей. Маршрутизаторы нижнего уровня обычно служат для связи с определенными конкретными частями сети, ее сегментами. Маршрутизаторы верхнего уровня образуют особую часть сети, называемую магистральной (опорной). Маршрутизаторы магистральной сети передают пакеты между сетями нижнего уровня;
  - *иерархическая* структура больших и сложных сетей способствует упрощению управления сетью, облегчает изоляцию сегментов сети и т. д. Кроме того, преимуществом иерархической маршрутизации является то, что она имитирует организацию предприятия и, как следствие, хорошо поддерживает его схему трафика;
- централизованными, распределенными или локализованными:
  - в *централизованных* алгоритмах маршрутизации выбор всех маршрутов осуществляется в центральном узле. Вся информация, касающаяся топологических изменений, передается в этот специальный узел, называемый центром управления сети. Центр хранит в памяти маршрутные таблицы всей системы и обновляет их после получения новой информации о топологических изменениях. После этого он приступает к передаче информации, необходимой для выполнения локальных операций по маршрутизации, всем узлам, которым эта информация адресована;
  - в *распределенных* алгоритмах вычисление маршрутов распределяется среди узлов сети, если необходимо, при помощи обмена информацией. В каждом узле что-то вычисляется, и результаты передаются в другие узлы;
  - в *локализованном* алгоритме каждому узлу надо знать о текущей топологии сети и вычислять маршруты ко всем возможным узлам назначения, основываясь на этой информации. Для того чтобы располагать самой свежей информацией, все узлы сети передают сообщения об их связности соседним узлам и так по цепочке;
- одноадресными или групповыми:
  - *одноадресные* алгоритмы маршрутизации строят один или несколько маршрутов к одному получателю;
  - *многоадресные* алгоритмы способны осуществить передачу информации многим получателям одновременно.

Когда маршрутизатор получает пакет, он выясняет пункт его назначения и определяет, по какому маршруту его отправить. Обычно маршрутизаторы

хранят данные о нескольких возможных маршрутах. Решение о маршрутизации зависит от нескольких факторов, в том числе от:

- применяемой системы измерения длины маршрута или метрики маршрута;
- основного алгоритма, используемого протоколом;
- топологии сети.

Классификация протоколов маршрутизации

Существует три основные группы протоколов маршрутизации. Деление на группы определяется типом реализуемого алгоритма определения оптимального маршрута.

- *Векторные протоколы* (известные также как алгоритмы Бэллмана-Форда) являются самыми простыми и самыми распространенными. Свое название этот тип протокола получил от способа обмена информацией. Маршрутизатор с определенной периодичностью копирует адреса получателей и метрику из своей таблицы маршрутизации и помещает эту информацию в рассылаемые соседям сообщения об обновлении. Соседние маршрутизаторы сверяют полученные данные со своими собственными таблицами маршрутизации и вносят необходимые изменения. После этого они сами рассылают сообщения об обновлении. Таким образом, каждый маршрутизатор владеет информацией о маршрутах во всей сети. При очевидной простоте алгоритма говорить о полной его надежности нельзя. Он может работать наиболее эффективно только в небольших сетях. Это связано с тем, что крупные сети не могут функционировать без периодического обмена сообщениями для описания сетевой топологии, и, к сожалению, большинство из них избыточны. Как следствие, возникают проблемы при выходе каналов связи из строя из-за того, что несуществующие маршруты могут оставаться в таблице маршрутизации в течение длительного времени. Трафик, направленный по такому маршруту, не достигнет своего адресата. В данную группу входит протокол RIP (Routing Information Protocol).
- *Протоколы состояния связей (канала)* впервые были предложены в 1970 году Дейкстрой. Они значительно сложнее, чем векторные протоколы. Вместо рассылки соседям содержимого своих таблиц маршрутизации, каждый маршрутизатор осуществляет широковещательную рассылку списка маршрутизаторов, с которыми он имеет непосредственную связь, и списка напрямую подключенных к нему локальных сетей. Эта информация является частью данных о состоянии связей и рассылается в специальных сообщениях. Кроме того, маршрутизатор рассылает сообщения о состоянии связей только в случае изменения информации о них или по истечении заданного интервала времени. Протоколы состояния связей трудны в реализации и нуждаются в значительном объеме

памяти для хранения информации о состоянии каналов. Примерами этих протоколов являются OSPF (Open Shortest Path First) и IS-IS (Intermediate System to Intermediate System).

- К третьей группе протоколов относятся *протоколы политики (правил) маршрутизации*. Они наиболее эффективно решают задачу доставки получателю информации по разрешенным путям. Эта категория протоколов используется в Интернете и позволяет операторам получать информацию о маршрутизации от соседних операторов на основании специальных критериев. Алгоритмы, используемые для политики маршрутизации, опираются на алгоритмы длины вектора, но информация о показателях путей базируется на списке операторов магистрали. Примерами протоколов данной категории могут служить BGP (Border Gateway Protocol) и EGP (Exterior Gateway Protocol).

Протокол вектора расстояний

Протокол вектора расстояний (Routing Information Protocol — RIP) относится к векторным протоколам. Основное преимущество алгоритма вектора расстояний — его простота. Действительно, в процессе работы маршрутизатор общается только с соседями, периодически обмениваясь с ними копиями своих таблиц маршрутизации. Получив информацию о возможных маршрутах от всех соседних узлов, маршрутизатор выбирает путь с наименьшей стоимостью и вносит его в свою таблицу.

Протокол вектора расстояний поддерживает только самые лучшие маршруты к пункту назначения. Если новая информация обеспечивает лучший маршрут, то она заменяет старую маршрутную информацию. Когда имеют место изменения в топологии сети, то они отражаются в сообщениях о корректировке маршрутизации. Например, когда какой-нибудь маршрутизатор обнаруживает отказ одного из каналов или другого роутера, он повторно вычисляет свои маршруты и отправляет сообщения о корректировке маршрутизации.

Достоинство этого элегантного алгоритма — быстрая реакция на хорошие новости (появление в сети нового маршрутизатора), а недостаток — очень медленная реакция на плохие новости (исчезновение одного из соседей).

За этот недостаток отвечает проблема, получившая название проблемы возрастания до бесконечности (count-to-infinity problem). Даже если узел узнал об исчезновении соседа и установил путь до него в таблице маршрутизации равным бесконечности, он тут же будет обманут другим своим соседом, который не знает о таком исчезновении и имеет конечный путь до того узла. Эта проблема является основной причиной того, что протокол вектора расстояний считает маршрут длиной более 15 транзитных узлов бесконечным. Такой путь будет немедленно удален из таблицы маршрутизации.

Для предотвращения образования ложных маршрутов предлагаются несколько методов.

- Метод *расщепления горизонта* заключается в том, что если известно, что путь до узла X лежит через соседний узел Y, то узлу Y не надо посылать объявления маршрута до X. Однако даже при минимальном усложнении топологии правило расщепления горизонта перестает действовать.
- Правило *отказа от приема* запрещает маршрутизатору, получившему сообщение об отказе узла, принимать объявления маршрута до этого узла в течение некоторого времени.
- Правило *принудительных объявлений* заключается в том, что, узнав об изменении метрики маршрута, маршрутизатор обязан немедленно сообщить об этом соседям.
- Метод *корректировки отмены маршрута* ("отравленного маршрута" — route-poisoning) расценивает значительно выросшую стоимость маршрута как признак образования петли. Такой маршрут удаляется из таблицы маршрутизации. Какое изменение стоимости маршрута понимать как "значительное", зависит от администратора.

Протокол вектора расстояний использует механизм принудительных объявлений, а также функцию временного отказа от приема сообщений для обеспечения большей стабильности работы в условиях изменяющейся топологии. Цена за такую стабильность — увеличение времени определения новых маршрутов, т. к., заблокировав изменение некоторого маршрута вследствие отказа какого-либо узла из опасения "дезинформации" со стороны соседей, маршрутизатор отбрасывает и корректные объявления. Многие реализации протоколов позволяют отключить функцию отказа от приема сообщений.

В этом случае из-за распространения ложной информации петли будут возникать чаще, но эффективность работы сети может повыситься.

Протокол выбора кратчайшего пути

Протокол выбора кратчайшего пути (Open Shortest Path First Protocol — OSPF) определяет принципы работы, которые в упрощенной форме можно сформулировать в виде пяти несложных правил. Итак, каждый маршрутизатор в сети должен:

- узнать всю информацию о топологии сети, измерить метрики каналов, соединяющих собственные физические интерфейсы с соседями, и далее с помощью алгоритма Дейкстры вычислить кратчайшие пути ко всем остальным узлам и внести полученные результаты в таблицу маршрутизации;
- узнать стоимость пути до каждого из соседей (т. е. узнать о состоянии каналов). Для этого он рассылает через все свои физические интерфейсы специальные пакеты с приветствием. Получив такой пакет, соседний узел должен ответить, сообщив данные о себе. Узнав данные о соседях, маршрутизатор принимается за второй пункт данной программы — тестирование каналов связи с целью выяснения метрики каждого канала.

Под метрикой может пониматься пропускная способность, время задержки, надежность (количество ошибок на единицу переданной информации), загрузка канала. Задержку канала можно определить, послав специальный пакет, который принимающая сторона должна немедленно отправить обратно. Разделив время отклика пополам, маршрутизатор вычисляет приблизительную величину задержки канала;

- подготовить пакет-объявление, содержащий полученную информацию;
- разослать этот пакет всем соседям. В процессе работы маршрутизатора пакеты с объявлениями маршрутов рассылаются обычно лишь в случае каких-либо изменений в сети. В остальное же время протоколы состояния канала молчат и не загружают каналы служебной информацией, лишь изредка обмениваясь небольшими пакетами с приветствием. Обмен информацией осуществляется с помощью веерной рассылки, т. е., получив пакет, маршрутизатор сохраняет копию в своей базе данных и посылает пакет дальше всем остальным соседям;
- построить дерево кратчайших расстояний до всех остальных маршрутизаторов ("карту" сети). Для этого создается ориентированный граф, отражающий топологию сети. Имея его в памяти, маршрутизатор применяет алгоритм Дейкстры для выбора пути с наименьшей суммарной стоимостью до каждой из вершин графа (т. е. до каждого узла сети). По результатам этих вычислений и строится таблица маршрутизации, используемая далее при переключении трафика.

Узким местом такого подхода является необходимость обязательной синхронизации баз данных всех маршрутизаторов в пределах автономной системы. Если разные узлы будут по-разному представлять себе топологию сети, с которой они работают, то это приведет к образованию петель и к другим проблемам.

Бесклассовая междоменная маршрутизация

Маршрут может относиться к полному IP-адресу, т. е. всего к одной машине. Однако обычно маршрутизация осуществляется на уровне сетей с использованием лишь части адреса пункта назначения. В прошлом межсетевая маршрутизация выполнялась на основе классов адресов, а подсети использовались только в пределах внутреннего шлюза. Это приводило к необходимости отдельного маршрута для сетей класса C, из-за чего пропускная способность базовой магистрали падала.

Схема бесклассовой доменной маршрутизации (Classless Inter-Domain Routing — CIDR) предлагает определять общие маршруты, которые обслуживали бы несколько сетей одновременно [Немет, Снайдер, Сибасс, Хейн 1999]. CIDR скрывает младшие биты номера сети и тем самым группирует несколько сетей в один маршрут. Эту схему, сформулированную в 1993 году, называют также *схемой организации надсетей.*

7.2.3.5. Формирование сети

Процесс формирования сети состоит из нескольких этапов, таких как:

- определение физической и логической топологии сети;
- установка сетевых аппаратных средств;
- назначение IP-адресов узлам сети;
- настройка всех основных машин на конфигурирование сетевых интерфейсов;
- настройка демонов маршрутизации, определение статических маршрутов.

Сетевая информационная служба (Network Information Services — NIS) — это распределенная база данных для хранения информации. Она появилась в ответ на проблему необходимости наличия единого информационного центра, поднимавшуюся системными администраторами. Выделенный узел хранит большое количество информации, в том числе:

- о текущем количестве узлов;
- о параметрах пользователей;
- о группах пользователей;
- о временной зоне.

Может существовать резервный сервер, с которым основной периодически обменивается информацией.

7.2.3.6. Средства коммуникации высокого уровня

Средства коммуникации высокого уровня изолируют приложение от специфики сетевого взаимодействия, включающего создание коммуникационных узлов, установление и завершение связи.

Удаленный вызов процедур

Удаленный вызов процедур (Remove Procedure Call — RPC) основан на том, что процесс, исполняющийся на одном компьютере, запускает процесс на удаленном компьютере. Фактически осуществляется вызов процедуры, которая реально находится и поддерживается на другом компьютере. Удаленный вызов внешне очень похож на локальный вызов, но в нем существенно отличаются действия по передаче параметров и получению результата. Действия выполняются с помощью передач информационных пакетов по сети и подробно могут быть описаны следующим образом [Робачевский 1997].

- Программа-клиент производит локальный вызов процедуры, называемой "заглушкой", при этом клиенту кажется, что, вызывая "заглушку", он на самом деле вызывает процедуру сервера. В действительности, задача

"заглушки" — принять аргументы, адресуемые вызываемой процедуре, преобразовать их в некоторый формат и сформировать сетевой запрос.

- Сетевой запрос пересылается по сети на удаленную систему, при этом, например, используется стек протоколов TCP/IP.
- На сервере все происходит в обратном порядке: "заглушка" сервера ожидает запрос и при его получении извлекает из него параметры.
- "Заглушка" сервера выполняет вызов настоящей процедуры (которой адресован запрос клиента), передавая ей нужные параметры.
- После выполнения процедуры при передаче результатов ее работы управление опять возвращается в "заглушку" сервера, которая формирует сообщение-отклик.
- Отклик передается клиенту.
- Отклик принимается "заглушкой" клиента, которая извлекает необходимые данные и передает их программе. Процесс завершен.

Фактически механизм заглушек составляет ядро системы удаленного вызова процедур. Удаленный вызов процедур передает только данные по значению (это серьезное ограничение). Более сложные среды распределенного программирования лишены подобного ограничения (например, среда CORBA). Прежде чем клиент сможет вызвать удаленную процедуру, необходимо связать его с удаленной системой, которая располагает требуемым сервером. Задача связывания состоит из нахождения удаленного узла с требуемым сервером и необходимого серверного процесса на данном узле. Укажем ряд проблем, связанных с данным средством:

- проблемы семантики вызова (например, невозможность установить, когда будет выполнена процедура и выполнится ли она вообще);
- проблемы представления данных (например, различное представление символов и вещественных чисел в различных архитектурах).

Коммуникации в группах

Основные проблемы групповой коммуникации заключаются в необходимости упорядочивания и синхронизации поступления сообщений. Даже небольшие задержки по времени в доступе до дальних узлов могут привести к нарушению порядка поступления сообщений на ряде узлов. Одним из удачных средств коммуникаций в группах является система ISIS. Это надстройка над Unix, содержащая ряд коммуникационных примитивов. В ней реализована совокупность алгоритмов, базирующаяся на идее векторов из `n` компонентов (по количеству процессов), управление которыми осуществляется системой. В начале работы каждый процесс получает пустой вектор. `i`-й компонент вектора является номером последнего сообщения, полученного в текущей последовательности от процесса под номером `i`. Рассмотрим, как эти векторы используются на практике (рис. 7.12).

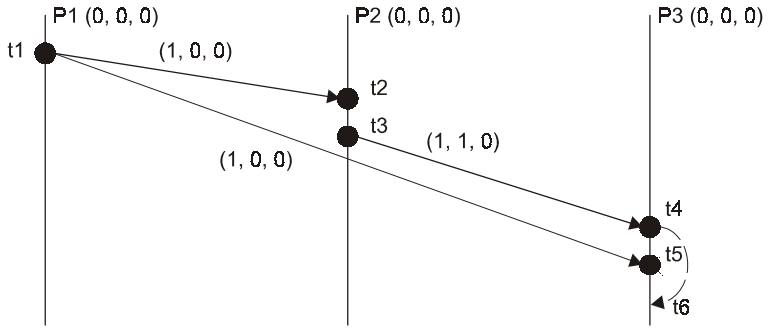


Рис. 7.12. Коммуникации в группах

Пусть процесс `P1` отправил в момент времени `t1` сообщение всем другим процессам. Процесс `P2`, получив сообщение от `P1` в момент времени `t2`, сам посылает остальным процессам сообщение в момент времени `t3`. Может оказаться так, что процесс `P3` получит сообщение от процесса `P2` в момент времени `t4`, т. е. раньше, чем от `P1`. Третий процесс анализирует вектор, пришедший от второго процесса `(1, 1, 0)`, сравнивая со своим текущим `(0, 0, 0)`. Обнаружив факт неполучения сообщения от первого процесса, он откладывает прием сообщения от второго процесса. После получения сообщения от процесса `P1` в момент времени `t5`, процесс `P3` примет отложенное сообщение в момент времени `t6`.

7.2.4. Синхронизация процессов в распределенных системах

7.2.4.1. Основные подходы к синхронизации

В традиционных архитектурах синхронизация выполнялась через общий для всех процессов участок. Процессы на разных узлах борются за критический участок. Синхронизация в распределенных системах использует децентрализованные алгоритмы и становится гораздо более сложной, чем в централизованных.

Такие алгоритмы обладают, как правило, следующими свойствами:

- относящаяся к делу информация распределена между множеством компьютеров;
- процессы принимают решения только на основе локальной информации;
- не должно быть единой критической точки, выход из которой приводил бы к краху алгоритма;
- не существует общих часов или другого источника точного глобального времени.

Первые три свойства говорят о том, что недопустимо собирать всю необходимую информацию для принятия решения в одном месте.

Алгоритм Лампорта

Алгоритм был предложен Лесли Лампортом (Leslie Lamport) в 1978 году. Основная идея заключалась в том, что абсолютной синхронизации не требуется, т. е. если два процесса не взаимодействуют, то единого времени и не требуется [Tanenbaum 1995].

Процессам достаточно иметь согласованное (логическое) время. Лампорт вводит специальный предикат "произошло до": `A–>B` означает, что "событие `A` произошло до события `B`".

Это соотношение очевидно в двух случаях:

- если оба события произошли в одном процессе;
- если событие `A` есть операция посылки сообщения одним процессом, а событие `B` — прием этого сообщения другим процессом.

Если два события случились в разных процессах, которые не обмениваются сообщениями, то соотношения `A–>B`, `B–>A` являются неверными, а события `A` и `B` — одновременными. Введем логическое время `C`, которое будем определять таким образом:

если `A–>B`, то `C(A) < C(B)`.

Часы `Ci` увеличивают свое значение с каждым событием в процессе `Pi` следующим образом: `Ci = Ci + d`, где `d > 0` и обычно `d = 1`. Если событие `a` есть посылка сообщения `m` процессом `Pi`, тогда в это сообщение вписывается временная метка `tm = Ci(a)`, и в момент получения этого сообщения процессом `Pj` его время корректируется таким образом: `Cj = max(Cj, tm + d)`. После `Ci` можно указывать номер процесса. Недостатки логического времени проявляются в системах реального времени.

Выборы координатора

Очень многие алгоритмы в распределенных системах требуют наличия *координатора* — одного из процессов, который должен выполнять специальные функции по инициации, координации или решению других задач.

В качестве координатора выбирается процесс с максимальным номером. Можно применять разные алгоритмы выборов, но если процедура началась, то она должна закончиться согласием всех процессов о выборе нового координатора.

Широковещательный алгоритм выбора

Один из процессов, обнаруживший отсутствие координатора, инициирует следующий алгоритм выбора:

- процесс посылает свое сообщение всем процессам с номером, бóльшим чем у него — "а не избрать ли нам координатора?";

- процессы с большими номерами отвечают инициатору, и его участие в выборах заканчивается. Эти процессы посылают сообщения "а не избрать ли нам координатора?" процессам с еще бóльшими номерами;
- в конце алгоритма останется один процесс, который не дождется ответа. Он и объявляет себя координатором.

Круговой алгоритм выбора

Круговой алгоритм использует физическое или логическое кольцо без маркера. При этом каждый процесс знает следующего за ним в круговом списке. Когда какой-либо из процессов обнаруживает отсутствие координатора, он посылает следующему за ним сообщение "а не избрать ли нам координатора?" и указывает свой номер. Если следующий не отвечает, то сообщение посылается процессу, идущему за ним. Каждый процесс добавляет в список свой номер. Рано или поздно список вернется процессу, инициировавшему выборы. Он посылает новое сообщение, в котором извещает о том, что он стал новым координатором, и указывает свой номер.

7.2.4.2. Взаимные исключения в распределенных системах

Централизованный алгоритм

Все процессы запрашивают у координатора разрешения на вход в критическую секцию и ждут разрешения. Координатор обслуживает запросы в порядке их поступления. Получив разрешение координатора, процесс входит в критическую секцию. Завершив работу, процесс докладывает об этом координатору.

Алгоритм с круговым маркером

Все процессы образуют логическое кольцо, при этом каждый знает, кто следует за ним. По кольцу циркулирует маркер, дающий право на критическую секцию. Получив маркер, процесс либо входит в критическую секцию, держа маркер у себя, либо перенаправляет маркер дальше, если ему не нужна критическая секция. Дважды подряд входить в критическую секцию, не отдавая маркер, нельзя.

Децентрализованный алгоритм на основе временных меток

Алгоритм требует глобального упорядочивания всех событий в системе по времени.

Когда процесс желает войти в критическую секцию, он посылает всем остальным процессам сообщение с именем критической секции, своим номером и текущим временем. После этого процесс ждет разрешения от всех процессов. Только после этого можно входить в критическую секцию.

Получив сообщение-запрос, процесс, в зависимости от своего состояния к указанной критической секции, действует одним из следующих способов:

- если получатель не находится внутри данной критической секции и не запрашивал разрешения на вход в нее, то он возвращает сообщение "разрешаю";
- если получатель находится в критической секции, то он не отвечает на запрос, но запоминает его;
- если получатель уже выдал запрос на вхождение в критическую секцию, но еще не вошел в нее (не дождался разрешения), то он сравнивает временные метки. Если у пришедшего запроса метка более ранняя, то он возвращает сообщение "разрешаю".

После выхода из критической секции процесс посылает сообщение "разрешаю" всем процессам, запросы от которых он запомнил, а затем все запомненные запросы стирает.

7.2.4.3. Высокоуровневые средства синхронизации

Атомарные транзакции

Атомарные транзакции — это высокоуровневые средства синхронизации, представляющие логические единицы работы. Логическая единица работы, как правило, является согласованием ряда операций. Система, поддерживающая процесс транзакции, гарантирует, что если во время выполнения некоторых операций произошла ошибка, то все обновления будут аннулированы. В результате транзакция либо полностью выполняется, либо полностью отменяется. Транзакции пришли в программирование из области деловых отношений, когда переговоры могут вернуться к своему начальному состоянию в любой момент до заключения соглашения.

Транзакции обеспечиваются администратором транзакций с помощью следующих операторов:

- `BEGIN_TRANSACTION` — объявляет начало транзакций;
- `END_TRANSACTION` — объявляет успешное завершение транзакции и утверждает ее;
- `ABORT_TRANSACTION` — указывает на неудачное завершение транзакции, в результате чего все изменения должны быть отменены, и восстанавливает начальные значения;
- `READ` — служит для чтения данных;
- `WRITE` — служит для записи данных.

Существуют два основных метода поддержки реализации транзакций.

- *Метод собственного пространства транзакции.* Все действия ведутся в собственном "частном" пространстве транзакции. Изменения попадут в

"реальное" пространство только в случае успешного завершения транзакции.

- *Метод протоколирования списка действий.* Все действия протоколируются в специальном списке, называемом "лист намерений". При изменении значения каждой переменной в список заносится имя переменной, а также ее начальное и конечное значения. Если транзакция завершается неудачно, то исходные значения переменных на начало транзакции легко восстановить по этому списку.

Транзакции обладают четырьмя важными свойствами [Дейт 2000].

- *Атомарность.* С точки зрения окружающего мира транзакция выглядит единой и неделимой. Выполняется либо все, либо ничего.
- *Постоянство (согласованность).* Если в системе есть некоторые инварианты и если они существовали до выполнения транзакции, то они должны остаться и после ее выполнения. Одно согласованное состояние переводится в другое. Обязательной поддержки согласованности в промежуточных точках нет.
- *Сериализация (изоляция).* Транзакции отделены одна от другой и преобразуются в последовательную форму. Если существует несколько транзакций, выполняющихся параллельно, то они должны выглядеть так, как если бы они выполнялись последовательно.
- *Прочность (долговечность).* Если транзакция завершена, то ее результаты надежно вписываются в состояние системы. Даже если в следующий момент произойдет сбой системы.

Двухфазный протокол утверждения

Двухфазный протокол утверждения важен в тех случаях, когда транзакция может взаимодействовать с несколькими независимыми администраторами ресурсов. Каждый из них руководит своим собственным набором восстанавливаемых ресурсов и поддерживает собственный файл регистрации. Для транзакции не имеет смысла выполнять оператор завершения отдельно для каждого из администраторов ресурсов. Вместо этого должна быть выполнена единая общесистемная команда `END_TRANSACTION` или `ABORT_TRANSACTION`. Процесс, запустивший транзакцию, объявляется координатором, остальные процессы — координируемыми. Координатор обеспечивает выполнение единой команды благодаря двухфазному протоколу утверждения.

Рассмотрим пример работы координатора. Пусть транзакция успешно завершена и ее нужно утвердить. Координатор осуществляет двухфазный процесс.

- Координатор записывает в протокол "транзакция готова к утверждению". Координируемые процессы, получив сообщение, записывают в свой протокол "готов к транзакции" и отсылают координатору сообщение

"готов". Координатор в конце первой фазы собирает все сообщения о готовности.

- Когда все ответы собраны, координатор записывает в протокол "транзакция утверждена" и рассылает всем процессам сообщение "транзакция утверждена". Координируемые процессы, получив сообщение, записывают в свой протокол "транзакция утверждена" и отсылают координатору "готов". Если все ответы во время первой фазы собрать не удалось или хотя бы один из ответов содержит отказ от транзакции, то координатор дает команду `ABORT_TRANSACTION`.

7.2.4.4. Тупики в распределенных системах

Тупики в распределенных системах подобны тупикам в централизованных системах, только их еще сложнее обнаруживать и предотвращать. Иногда выделяют особый вид тупиков в распределенных системах — коммуникационные тупики. Стратегии, применяемые при борьбе с тупиками в распределенных системах, таковы:

- *полное игнорирование проблемы* ("алгоритм страуса"). Этот подход в распределенных системах так же популярен, как и в централизованных;
- *обнаружение тупиков*. Для этого существует два основных метода:
  - централизованное обнаружение тупиков заключается в распространении идеи графа размещения ресурсов на распределенную систему. Координатор строит единый граф для всех узлов системы и на нем выполняет редукцию (приведение);
  - распределенное обнаружение тупиков. В этом методе инициирование задачи обнаружения тупика начинает процесс, который подозревает, что система находится в тупике. Он посылает тем процессам, которые держат нужный ему ресурс, сообщение из трех информационных полей. Первое поле содержит номер процесса, инициировавшего поиск, и оно не меняется при дальнейшей пересылке. Второе и третье поля содержат соответственно номер процесса, который зависит от ресурса, и номер процесса, который держит этот ресурс. Если, в конце концов, сообщение вернется к тому процессу, который инициировал поиск, то он приходит к выводу, что система зациклена и он должен предпринять действия по ликвидации тупика;
- *предотвращение тупиков* в распределенных системах базируется на идее упорядочивания процессов по их временным меткам. Допустимой будет ситуация, когда "старший" процесс ждет ресурсы, которые захватил "младший" процесс. В обратном случае "младший" процесс должен отказаться от ожидания.

Тупики в распределенных системах являются той областью исследования, где наука и практика резко расходятся с самого начала, и многие алгоритмы никогда не будут воплощены на практике.

7.2.4.5. Распределение процессоров в распределенных системах и планирование

Модели организации процессоров в распределенных системах

Процессоры в распределенных системах могут быть организованы следующим образом:

- *модель рабочих станций*. Это модель системы, состоящей из рабочих станций, объединенных в локальную сеть;
- *модель процессорного пула*. Данная модель состоит из массива процессоров (процессорного пула) и X-терминалов;
- *гибридная модель*, соединяющая в себе особенности двух предыдущих моделей.

Модель рабочих станций может включать рабочие станции нескольких категорий.

- Бездисковые. И для загрузки операционной системы, и для дальнейшей работы такая рабочая станция обращается к серверу. Особенности этой категории в низкой стоимости и достаточно легком сопровождении.
- С диском, используемым для хранения временных файлов и области подкачки страниц.
- С диском, используемым для хранения временных файлов, области подкачки страниц и системных файлов.
- С диском, используемым для хранения временных файлов, области подкачки страниц, системных файлов и кэшированных файлов (локальных копий для редактирования).
- С полноценным жестким диском.

Для работы с удаленным узлом может использоваться примитив `remote a.out` — запуск на простаивающем узле задачи `a.out`. При разработке алгоритмов работы в такой модели следует продумать решение трех вопросов.

- Как определить простаивающую рабочую станцию?
- Как сообщить системе, что данная рабочая станция теперь не простаивает?
- Если на занятой нашим заданием удаленной рабочей станции начал работать пользователь, то что следует делать?

Как правило, для решения данных вопросов в модели рабочей станции применяется идея координатора, организующего работу процессоров.

- Простаивающая рабочая станция должна сообщить координатору о том, что она не занята. Координатор поддерживает у себя информацию о незанятых рабочих станциях.

- Процесс, желающий воспользоваться простаивающей рабочей станцией, обращается к координатору.
- Если у координатора есть свободные рабочие станции, то он посылает заинтересованному процессу информацию о такой станции.
- Процесс обращается к свободной рабочей станции с просьбой выделить ему ресурсы. На каждой рабочей станции есть менеджер, управляющий распределением ресурсов.
- Менеджер рабочей станции обращается к координатору с просьбой удалить данную рабочую станцию из списка свободных.
- Процесс выполняет установку окружения на удаленной рабочей станции.
- Процесс запускает работу на удаленной рабочей станции.
- Процесс выполняет работу на удаленной рабочей станции.
- Процесс завершает работу на удаленной рабочей станции.
- Процесс посылает координатору сообщение о завершении им работы на удаленной станции.

В случае модели процессорного пула основной задачей является выстраивание процессов в очередь к этому пулу. Пусть `x` — количество задач, генерируемых в секунду пользователями, а `y` — количество задач, которое процессорный пул способен обработать в секунду. Леонард Клейнрок (Leonard Kleinrock) доказал, что среднее время между возникновением запроса и получением ответа на него может быть оценено как:

```
T = 1/(y − x).
```

Пусть есть `n` процессорных пулов. Тогда:

```
T′ = 1/(yn − xn) = T/n.
```

Данная формула показывает, что замена `n` небольших ресурсов одним большим, в `n` раз более мощным, позволит сократить время ответа в `n` раз.

Распределение процессов по процессорам

Наиболее интересной и распространенной моделью является модель рабочих станций. Далее будем считать, что все рабочие станции имеют одинаковую архитектуру. Однако они могут различаться как по быстродействию, так и по количеству памяти.

Различают два основных класса стратегий распределения.

- *Немигрирующие* стратегии, в которых однажды помещенный на рабочую станцию процесс больше не меняет это место.
- *Мигрирующие* стратегии, в которых передача процесса с одной рабочей станции на другую возможна несколько раз.

На чем может основываться идея распределения процессов? В ее основе может лежать:

- минимизация времени простоя процессоров;
- минимизация времени исполнения процессов.

Алгоритмы распределения могут быть сведены к пяти основным проектным решениям.

- Детерминированные (четко определенные) или эвристические алгоритмы. В случае детерминированных алгоритмов система должна вести себя все время одинаково, т. е. все о процессах должно быть известно заранее.
- Централизованная или распределенная реализация алгоритма. Централизованные алгоритмы учитывают присутствие координатора, с помощью которого легче принимать решения.
- Оптимальные или приемлемые алгоритмы. В оптимальных алгоритмах все построения и вычисления выполняются с огромной точностью, но при этом тратится большое количество времени.
- Локальная или глобальная политика переноса процессов.
- Инициируемое источником (т. е. рабочей станцией, на которой выполняется много процессов) или простаивающей рабочей станцией управление распределением процессов.

Реально существующие распределенные системы обычно используют эвристические распределенные приемлемые алгоритмы.

Некоторые реализационные особенности алгоритмов

Если имеется переполненная процессами рабочая станция, то часть процессов необходимо перенаправить на другие рабочие станции. Проблема заключается в том, чтобы выбрать кандидата. Существуют три основных подхода к его поиску.

- Узел, на который будет переведен процесс, определяется случайным образом. Далее возможны два варианта: либо новая рабочая станция слабо загружена и процесс начинает работу на ней, либо станция загружена сильно и тогда продолжается поиск нового узла. Для того чтобы количество пересылок не стало слишком большим, обычно вводят ограничения на их количество.
- Новый узел также выбирается случайным образом, но вместо пересылки туда процесса высылается только запрос. Если ответ отрицательный, то продолжаем искать незагруженный узел. Ограничение на число пересылок есть и в этом случае.
- Одновременно посылаются запросы ко всем узлам сети, а затем анализируются ответы. В результате анализа можно выбрать минимально загруженный узел.

Примеры алгоритмов распределения

Рассмотрим три различных алгоритма, демонстрирующих, как следует выделять процессоры в распределенных системах [Tanenbaum 1995].

- *Детерминированный алгоритм на графах.* Этот алгоритм в первую очередь используется в системах, где заранее известно число процессоров, объем памяти на каждой рабочей станции и величина ее загрузки. В качестве количественной характеристики берется связь процесса с другими процессами. Причем она учитывается только в том случае, если процессы находятся на разных рабочих станциях. Согласно этому алгоритму следует переносить процессы с одной рабочей станции на другую, чтобы уменьшить нагрузку.
- *Централизованный алгоритм.* На одном из узлов для каждой рабочей станции ведется анализ ее загруженности. В простейшем случае для каждого узла строится график. Если система не загружена, то график опускается вниз. При загрузке системы график поднимается вверх, причем движение осуществляется скачками по числу работающих в системе процессов.
- *Экономический алгоритм.* Компьютерная система рассматривается с точки зрения экономических моделей. Некоторый координатор собирает о рабочих станциях данные, включающие информацию о процессоре, памяти и других характеристиках. В результате определяется стоимость использования рабочих станций, выраженная некоторыми числами. Координатор принимает решение, опираясь на стоимость и некоторые дополнительные факторы, связанные с процессом (например, его цель).

В каждом узле возникает задача диспетчеризации, которая в подавляющем большинстве случаев решается локально. Некоторые проблемы могут возникнуть лишь в том случае, если группа сильно взаимодействующих процессов попадает на разные рабочие станции.

7.3. Память

Если ты что-то записал в компьютерной памяти, запомни, где ты это записал.
Лео Бейзер

Иерархия памяти современных компьютеров с точки зрения архитектуры была рассмотрена в *разд. 6.3.2.* Здесь мы познакомимся с тем, как операционная система работает с данным ресурсом.

7.3.1. Основная память

7.3.1.1. Привязка адресов

На рис. 7.13 изображен процесс отображения информации исходной программы в оперативную память. Больше всего нас будут интересовать две функции.

- *Функция именования*, которая однозначно отображает данное пользовательское имя в идентификатор информации, к которому относится имя. Эта функция реализуется обычно редактором связей.

- *Функция памяти*, которая однозначно отображает определение идентификатора в истинные номера ячеек памяти, в которых он будет находиться. Эта функция реализуется частью операционной системы, называемой загрузчиком.

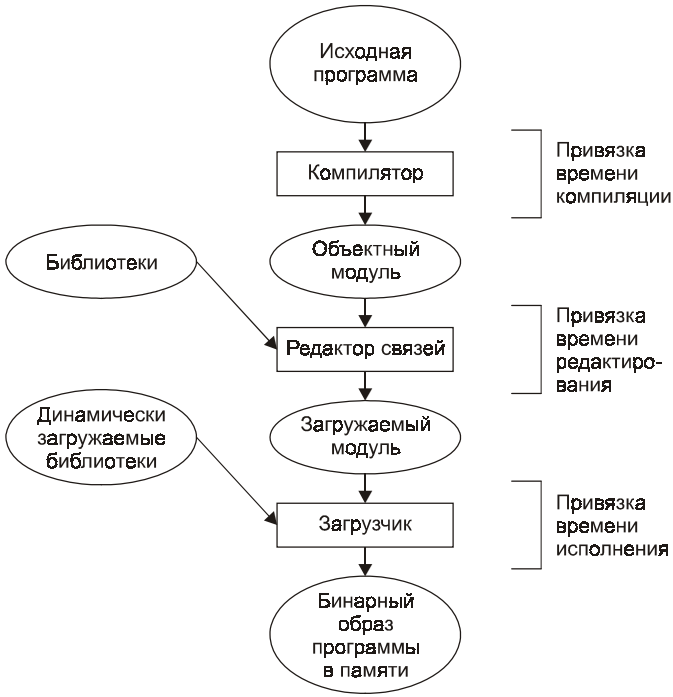


Рис. 7.13. Привязка адресов

Функция именования

Функцию именования реализует *редактор связей* — программа, объединяющая объектные модули в один модуль загрузки. Практически всегда редактор связей является частью операционной системы. Достаточно часто эта программа входит в состав компилятора, позволяя эффективным образом объединять объектные модули именно этого компилятора. Каждая операционная система имеет свои соглашения о связях, т. е. правила оформления структуры файла, который является результатом работы редактора связей.

Перечислим некоторые из проблем, возникающих перед редактором связей.

- Редакторы связей сталкиваются с проблемой установки соответствия между одноименными объектами в языках с блочной структурой.
- При трансляции отдельного модуля должна быть предусмотрена специальная таблица с информацией о глобальных объектах, которые используются в данном модуле, но не определены в нем.

- Редактор связей обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.

Функция памяти

Функция памяти выделяет переместимому модулю загрузки реальные ячейки памяти.

Распределение памяти может быть:

- *статическим*, когда привязка к конкретным ячейкам памяти выполняется либо до, либо во время загрузки модуля;
- *динамическим*, когда задание может перемещаться в памяти, допуская настройку и получение абсолютных значений ячеек памяти при каждом обращении к ним.

Существуют следующие основные способы реализации отображения памяти:

- *оверлеи* (overlay) — возможность расположить модули в памяти таким образом, чтобы один из них (корневой) постоянно находился там, а остальные попеременно загружались в ходе выполнения программы в одну и ту же область, сменяя и перекрывая друг друга;
- *свопинг* (swapping) — разрешение системе вводить в память и выводить из нее задания целиком;
- *поблочное отображение* — возможность группировать элементы информации в блоки (если блоки имеют одинаковый размер, то это страницы, если разный, то сегменты).

Как правило, все современные архитектуры имеют мощную аппаратную поддержку работы с блоками памяти. Это объясняется следующими факторами:

- во-первых, с помощью аппаратной поддержки легко убедиться, что мы пытаемся адресоваться именно в те участки, в которые разрешен доступ;
- во-вторых, имеются средства обеспечения быстрого доступа к нужному участку с помощью ассоциативной памяти, реализованной на высокоскоростном кэше. Такие средства обычно называют буфером быстрого преобразования адреса (Translation Look-aside Buffer — TLB). Во многих случаях эти средства играют решающую роль в обеспечении эффективности работы. Даже наличие 8- или 16-регистрового кэша позволяет поднять ее на 90 %.

Обычно используется так называемая сегментно-страничная организация памяти, в которой сегменты разбиваются на страницы фиксируемой длины.

7.3.1.2. Управление виртуальной памятью

Виртуальная организация памяти — это процесс расширения логической памяти за пределы физической.

Виртуальная память — это техника, позволяющая исполнять процессы, которые могут находиться в памяти не полностью.

Виртуальная память реализована во многих операционных системах. Однако есть операционные системы, не поддерживающие ее по разным причинам:

- MS-DOS — по причине распространенности на старых процессорах со слабой аппаратной поддержкой;
- операционные системы на архитектуре Cray — с целью экономии времени.

Виртуальная память решает две основные задачи.

- Возможность одновременной работы с несколькими приложениями.
- Возможность отсутствия программы в основной памяти целиком.

Конкретный механизм виртуальной памяти зависит от того, как реализуются три основные стратегии.

- Стратегия *размещения* (placement policy) определяет, в какое место основной памяти будет помещена подкачиваемая страница или сегмент.
- Стратегия *вталкивания* (fetch policy) определяет, в какой момент времени страница или сегмент должна быть помещена в основную память.
- Стратегия *вытеснения* (replacement policy) определяет, какую из страниц или сегментов нужно удалить из основной памяти, чтобы поместить на ее место новую.

Стратегия размещения

В случае страничной организации памяти стратегия размещения реализуется тривиально, поскольку все участки памяти имеют один и тот же размер. В случае сегментной организации памяти выделяют два подхода.

- *Ведение списка свободной памяти.* Список можно организовать по возрастанию адресов или размера фрагментов. Известной стратегией ведения списков является метод близнецов в случае блоков размером 2 в некоторой степени.
- *Сборка мусора*, представляющая собой уплотнение памяти. Сборка мусора имеет достаточно много отрицательных моментов, например при ней приходится приостанавливать выполнение текущих заданий.

Стратегия вталкивания

Существует две основные стратегии вталкивания (подкачек).

- *Вталкивание по запросу*, осуществляемое в тот момент, когда требуется отсутствующая страница.
- *Вталкивание с опережением.* Оно может быть применено, если имеется возможность предсказать поведение программы. Это может быть сделано

в очень редких случаях (например, при работе с большим массивом, который обрабатывается последовательно).

В том случае, когда нужна отсутствующая в памяти страница, операционная система генерирует прерывание. Далее происходит обращение к менеджеру виртуальной памяти. Он считывает с жесткого диска нужную страницу, записывает ее в свободный участок памяти, корректирует таблицу страниц и дает процессору команду на повторное выполнение действия.

Стратегия вытеснения

Для определения вытесняемой из основной памяти страницы используются алгоритмы, приведенные в книге [Цикритизис, Бернстайн 1977].

- *Принцип оптимальности* (принцип Биледи). Следует вытеснять ту страницу, к которой не будет обращения в течение наиболее длительного времени. Этот принцип нереализуем на практике, но может быть использован как эталон при оценке других алгоритмов.
- *Вытеснение случайно выбранной страницы*. Недостаток метода в слепом везении, достоинство — в отсутствии дискриминации.
- *Вытеснение первой загруженной страницы*. Достоинство метода — в его легкой реализации, недостаток — в неэффективности при большой загрузке системы. Существует аномалия данного метода, когда определенные последовательности обращений к страницам приводят к увеличению количества прерываний по отсутствию страницы в памяти при увеличении количества свободных фреймов.
- *Вытеснение дольше всего неиспользуемой страницы*. Этот метод требует дополнительной аппаратной или программной поддержки для хранения последнего времени доступа.
- *Вытеснение реже всего используемой страницы*. Этот метод требует дополнительной аппаратной или программной поддержки для хранения числа обращений.
- *Вытеснение не используемой в последнее время страницы*. Поддержка заключается в хранении двух битов. Первый содержит признак обращения к странице, второй — признак модификации страницы. Первоначально все биты сбрасываются в ноль. Существует четыре комбинации битов, которые можно выстроить по приоритету. Сначала, например, удалять страницы с комбинацией 0-0, потом с 0-1, 1-0 и, наконец, 1-1.
- *Принцип локальности. Локальность* — свойство, заключающееся в том, что распределение запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации, как временной, так и пространственной.
  - *Временная локальность* заключается в том, что если к некоторым ячейкам памяти недавно были обращения, то с высокой вероятностью будут обращения к ним и в будущем.

- *Пространственная локальность* заключается в том, что если к некоторым ячейкам памяти недавно были обращения, то с высокой вероятностью можно ожидать обращения и к ближайшим к ним ячейкам.

Рабочее множество — это набор страниц, к которым процесс активно обращается. Для того чтобы программа выполнялась эффективно, необходимо, чтобы ее рабочее множество находилось в основной памяти.

7.3.1.3. Распределенная общая память

С момента возникновения идеи распределенных вычислений явно предполагалось, что программы на машинах без физической общей памяти исполняются в разных адресных пространствах. Только в 1986 году Кан Ли (Kai Li) создал концепцию *распределенной общей памяти* — РОП (Distributed Shared Memory — DSM). Основная идея заключается в том, что набор рабочих станций в сети разделяет единое страничное виртуальное адресное пространство.

Достоинства систем с распределенной общей памятью в том, что в них могут исполняться программы, написанные для мультипроцессорных систем. Такие системы легко масштабируются. Недостатком является достаточно низкая производительность, связанная с накладными расходами на выполнение коммуникационных протоколов (trashing).

Путь оптимизации систем с распределенной общей памятью может заключаться в разделении только тех данных, которые требуются для записи, а данные для чтения не разделяются (следовательно, эти страницы не перегоняются по сети).

Типы архитектур и соответствующие им структуры, поддерживающие распределенную общую память, приведены на рис. 7.14 [Tanenbaum 1995].

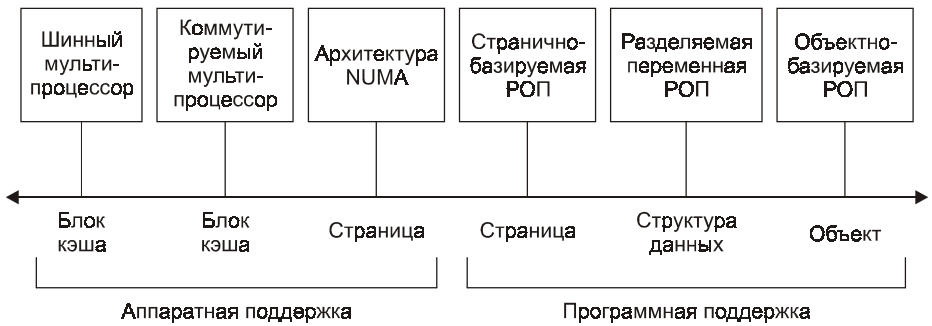


Рис. 7.14. Распределенная общая память

Алгоритмы реализации распределенной общей памяти

Существуют четыре основных алгоритма [Tanenbaum 1995].

- Алгоритм *с центральным сервером*. Все разделяемые данные поддерживает центральный сервер. Клиенты обращаются к нему с запросами. Если

данные нужны для чтения, то они посылаются клиентам. Если для записи, то сервер их корректирует и посылает клиентам в ответ квитанции, подтверждающие модификацию.

- *Миграционный* алгоритм. Основное отличие данного алгоритма от предыдущего заключается в том, что отсутствует централизованная поддержка данных сервером. Разделяемые данные пересылаются (мигрируют) на тот узел, который их запросил. Алгоритм предусматривает возможность задать время, в течение которого страница насильственно удерживается в узле для того, чтобы можно было выполнить несколько обращений к ней до миграции в другой узел.
- Алгоритм *разделения для чтения*. Разделяются концепции чтения и записи. Страницы для чтения могут храниться хоть на каждом узле. Страницы для записи должны управляться по предыдущему алгоритму. Производительность алгоритма повышается за счет возможности одновременного доступа по чтению. Запись данных требует больших затрат для уничтожения всех устаревших копий блока данных или их коррекции.
- Алгоритм *полного размножения*. Этот алгоритм — расширение предыдущего. Он реализует протокол многих читателей и многих писателей. Поскольку много узлов могут писать данные параллельно, требуется контролировать доступ к ним для поддержания согласованности.

Все перечисленные алгоритмы недостаточно эффективны. Для повышения эффективности следует менять не алгоритмы работы с памятью, а семантику (способ) работы с ней.

Модель консистентности (логичности)

Модель консистентности представляет собой некий договор между программами и памятью, в котором указано, что при соблюдении программой определенных правил работы, содержимое модуля памяти будет корректно, а если требования к программе будут нарушены, то память не гарантирует правильность выполнения операций "чтение — запись". Рассмотрим основные типы консистентности.

- *Строгая консистентность*. Операция "чтение ячейки памяти с адресом x" должна возвращать значение, записанное самой последней операцией "запись" с адресом x. В системе со строгой консистентностью должно присутствовать физическое (реальное) время, что невозможно. Для большинства систем это нереализуемо.
- *Последовательная консистентность*. Впервые она была определена Лампортом в 1979 году. Результат выполнения должен быть тот же, как если бы инструкции операторов всех процессов выполнялись в некоторой последовательности, определяемой программой для этого процессора. При параллельном выполнении все процессы должны видеть одну и ту же последовательность записей в память (разрешаются запаздывания для чтения).

- *Причинная консистентность.* Не требует, чтобы все процессы видели одну и ту же последовательность записей в память, проводя различие между потенциально-зависимыми (в одни и те же ячейки) и потенциально-независимыми (в различные ячейки) операциями записи.
- *Водопроводная консистентность.* Операции записи, выполняемые одним процессором, видны всем остальным процедурам в том порядке, в котором они выполнялись. Операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке.
- *Слабая консистентность.* Эту консистентность определяют три правила:
  - доступ к синхронизационным переменным определяется моделью последовательной консистентности;
  - доступ к синхронизационным переменным запрещен, пока не выполнены все предыдущие операции записи;
  - доступ к данным по записи или чтению запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным для данного (локального) процессора.

7.3.2. Внешняя память

7.3.2.1. Управление внешней памятью

Единственной мерой времени является память.

Владислав Гжегорчик

В управлении внешней памятью выделяют две подзадачи.

- Управление файловой системой.
- Управление устройствами ввода-вывода, представляющими часть вычислительной системы, действия которой ориентированы на обмен информацией с центральным процессором.

Отметим, что схема управления файлами образует некоторую иерархию.

- Уровень системы управления базами данных. Это высший уровень иерархии.
- Методы доступа.
- Логическая система управления файлами. На этом уровне выполняются такие логические команды, как, например, открыть файл.
- Базисная система управления файлами. Это уровень работы с сегментами данных.
- Система ввода-вывода. Например, прочитать физический блок памяти.
- Уровень аппаратуры. Это действия, ориентированные на устройства. Примером может быть команда нижнего уровня — перемотать ленту.

7.3.2.2. Файлы и файловые системы

За это время испортил массу чистых файлов.
Рома Воронежский

Файл — это поименованная упорядоченная совокупность данных. Файл обычно имеет следующий минимум атрибутов:

- имя файла. Это строка символов, поддерживающая информацию в виде, удобном для восприятия человеком;
- тип файла. Данная информация требуется многим утилитам и системам для первичной идентификации файла;
- местонахождение. Это информация об устройстве, на котором файл расположен, и полный путь к нему в рамках этого устройства;
- размер файла;
- информация о защите файла от чтения, записи и исполнения;
- информация о пользователе, создавшем файл, включая дату и время создания;
- информация о пользователе, который последним изменял файл, включая дату и время модификации.

Операции, выполняемые с файлами, достаточно очевидны:

- создание файла;
- запись в файл;
- чтение из файла;
- позиционирование в файле;
- удаление файла;
- удаление некоторой части файла, начиная с определенной позиции.

Информация, связанная с открытым файлом, такова:

- описатель файла;
- количество открытий этого файла;
- информация о местоположении файла на диске. При этом фактически копия открытого файла (или его части) хранится в основной памяти.

Файл является частью следующей иерархии объединения данных:

- комбинация битов;
- байт;
- поле — группа взаимосвязанных байтов;
- запись — группа взаимосвязанных полей. Записи могут быть физические (блоки) и логические (рассматриваемые как единое целое с точки зрения пользователя);
- файл — группа взаимосвязанных записей.

Организация файлов — это способ расположения записей файлов во внешней памяти. Различают пять основных способов организации.

- Последовательная, в которой записи следуют одна за другой. Примером может служить организация записей на магнитной ленте.
- Библиотечная, в которой блоки объединены в разделы.
- Прямая (произвольная), в которой доступ к записям осуществляется по их физическим адресам.
- Индексно-последовательная, в которой используются ключи для поиска физических записей.
- Виртуально-последовательная, в которой записи упорядочены по ключу и состоят из справочника (как правило, B-дерева) и области данных.

Метод доступа — это сочетание типов файлов и стандартных программных средств, обеспечивающих определенный режим передачи данных между файлом и основной памятью. Существует два метода.

- *Метод доступа с очередями.* Применяется, когда последовательность обработки записей можно предвидеть (например, при последовательном доступе).
- *Базисный метод доступа.* Этот метод доступа применяют в том случае, когда последовательность обработки предвидеть нельзя.

7.3.2.3. Распределенные файловые системы

При создании распределенных файловых систем обращают внимание на две главные задачи [Tanenbaum 1995].

- *Сетевая прозрачность*, которая заключается в обеспечении тех же возможностей доступа к файлам, как и в централизованных системах.
- *Высокая доступность*, заключающаяся в том, что ошибки и системные сбои не должны приводить к проблемам доступа к файлам.

Файловый сервис — интерфейс с файловой системой, то, что предоставляет файловая система.

Файловый сервер — это процесс, который представляет файловый сервис.

Пользователь не должен знать, сколько в системе файловых серверов и где они расположены. В системе могут функционировать разные процессы (с разных операционных систем), следовательно, файловый сервис должен уметь работать со всеми.

Архитектура распределенных файловых систем

Распределенная файловая система обычно имеет два существенно отличающихся компонента.

- *Файловый сервис.* Модели, на которых может основываться файловый сервис, таковы:
  - модель "загрузки-разгрузки". В этом случае осуществляется пересылка файла клиенту целиком;

- модель удаленного доступа. Реализуется без пересылки файла клиенту;

❒ *Сервис каталогов (директорий)*, обеспечивающий операции создания и удаления каталогов, именования файлов, их переименования и перемещения.

Ключевые решения, которые необходимо принять при разработке распределенных файловых систем, определяются ответами на следующие вопросы.

❒ Должны или не должны все процессы видеть иерархию каталогов одинаково?

❒ Должен ли быть единый корневой каталог?

Существуют две формы прозрачности именования.

❒ Прозрачность расположения, определяющая, как легко мы можем обратиться к файлу.

❒ Прозрачность миграции, когда изменение расположения файла не требует изменения имени.

Подходы к именованию файлов таковы:

❒ процессы видят файл как имя машины и путь к файлу на ней;

❒ монтирование удаленных файловых систем в локальную иерархию файлов;

❒ все процессы видят все файлы одинаково (для реализации нужен мощный механизм).

Большинство систем используют ту или иную форму двухуровневого именования: файлы имеют символическое имя, которое видит пользователь, и внутреннее двоичное имя, которое используется самой системой.

Семантика разделения файлов такова:

❒ семантика однопроцессорного компьютера. Если за операцией записи следует операция чтения, то результат определяется последней операцией записи;

❒ семантика сессий. Изменения открытого файла видит только тот процесс, который эти изменения проводит, и лишь после закрытия файла изменения становятся доступными всем остальным процессам;

❒ семантика транзакций. Монополизация доступа, а после возврата — обновление.

Реализация распределенных файловых систем

Рассмотрим различные аспекты реализации распределенных файловых систем [Tanenbaum 1995].

❒ *Использование файлов.* Для разработчика главное — понять, как система будет использовать файлы. Приведем некоторые данные статистики.

- Большинство пользовательских файлов имеет размер менее 10 Кбайт. Средний размер файла равен 4 Кбайт. Вывод может быть таким, что следует осуществлять перекачку файлов целиком.

- Чтение встречается чаще, чем запись. Вывод: следует активно применять кэширование.
- Чтение и запись осуществляются последовательно, произвольный доступ очень редок. Вывод: можно вести упреждающее кэширование.
- Большинство файлов имеет короткое время жизни. Вывод: временные файлы надо создавать на машине клиента.
- Лишь небольшое количество файлов разделяется процессами. Вывод: кэширование надо делать на машине клиента и применять семантику сессий.
- В системах существуют различные классы файлов с различными свойствами. Вывод: для разных классов файлов в системе могут поддерживаться различные механизмы.

- *Структура системы.* Для ее определения должны быть получены ответы на три вопроса.
  - Есть ли разница между клиентом и сервером? Здесь есть два варианта: либо любая машина может предоставить файловый сервис, либо есть специализированные средства сервера.
  - Следует ли объединять файловый сервер и сервер каталогов? Разумным может быть следующее разделение: можно иметь различные серверы каталогов (например, для операционных систем Unix и DOS) и один файловый сервер.
  - Должен ли сервер хранить у себя информацию о клиентах? В зависимости от принятого решения в системе будут либо серверы с состоянием, либо без состояния.

- *Кэширование.* В клиент-серверной схеме с памятью и диском имеются четыре места для хранения файлов или их частей.
  - Диск сервера. Данное место удобно тем, что здесь легко поддержать консистентность и нет размножения файлов, но при этом теряется эффективность и возникают проблемы передачи данных.
  - Память сервера. При этом возникает дополнительный вопрос — помещать файлы в память целиком или частями.
  - Диск клиента. Это место не дает преимуществ перед первыми двумя, но в несколько раз усложняется работа системы.
  - Память клиента. Это место является наиболее разумным для хранения файлов или их частей.

Оценить способ и обосновать выбор можно лишь при учете характера приложений и данных о быстродействии процессоров, памяти и сети.

7.4. Драйверы устройств

Танк — специализированное периферийное устройство
для программ управления огнем.
Программистский фольклор

Драйвер — это программа операционной системы, обеспечивающая взаимодействие исполняемой программы с отдельным устройством и способствующая его удобному использованию. Существуют, например, драйверы дисплея, мыши, клавиатуры и т. п. Выделяют следующие классы драйверов.

- *Драйвер физического устройства*, позволяющий конкретному физическому устройству взаимодействовать с операционной системой. Если устройство входит в список совместимого оборудования, то драйвер такого устройства практически всегда входит в состав операционной системы. Тем не менее, драйверы обычно пишутся не компанией, создающей операционную систему, а компанией, разработавшей периферийное устройство. При установке нового устройства операционная система либо находит драйвер в своей поставке, либо просит пользователя установить драйвер (например, с диска, поставляемого с устройством, или с сайта разработчика в сети Интернет). Примером данного класса драйверов является драйвер мыши.
- *Драйвер логического устройства*, обслуживающий логические устройства, относящиеся к одному типу. Примером данного класса драйверов может быть драйвер логических дисков, на которые разбивается физический диск.
- *Драйвер виртуального устройства (псевдоустройства)*, предоставляющий доступ к виртуальным устройствам, имитируемым программным или аппаратным способом. Примером псевдоустройства является `/dev/null` в Unix.

Поскольку большинство операционных систем запрещают программам напрямую взаимодействовать с устройствами, то для программ драйвер становится единственным способом работы с устройством. В случае многозадачных операционных систем драйвер обеспечивает разделенный доступ к устройствам для различных программ.

Алгоритм работы драйвера поддерживается основной нитью, обработчиком прерывания и, возможно, несколькими дополнительными нитями, созданными обработчиком. Все эти нити исполняют работу конечного автомата, имеющего диаграмму состояний (таблицу последовательных переходов и выходов).

Работа с внешним устройством определяется протоколом, который состоит из анализа запроса, передачи команды устройству, ожидания прерывания по завершении этой команды, анализа результата операции и формирования ответа внешнему устройству. Многие запросы приводят к выполнению не одной, а целой последовательности команд.

Во многих операционных системах имеется два типа устройств: устройства ввода-вывода блоками и устройства неструктурированного или посимвольного ввода-вывода. Примером блочных устройств ввода-вывода являются диски и ленты. К устройствам посимвольного (побайтного) ввода-вывода относятся практически все остальные устройства, в том числе дисплеи, клавиатуры и сетевое оборудование. Устройства ввода-вывода блоками могут также иметь интерфейс работы посимвольного ввода-вывода. В последнем случае для работы с устройством существуют два драйвера: блочный и посимвольный.

Правильное управление устройствами ввода-вывода может резко повысить эффективность системы. В качестве примера обратимся к такому устройству как диск. Головка считывания-записи должна быть подведена к определенному цилиндру диска, а затем к нужному сектору цилиндра. Очевидно, что время подвода головки к цилиндру значительно выше, чем время перемещения головки к сектору при вращении диска, и основное внимание в алгоритмах следует обращать на него. Далее приведем лишь некоторые возможные алгоритмы перемещения головки.

- Обслуживание запросов в порядке поступления. Алгоритм эффективен только в случае очень маленькой загруженности устройства.
- Обслуживание сектора с наименьшим временем поиска первым. В случае данного подхода происходит дискриминация отдаленных секторов.
- Сканирование. Головка движется взад и вперед, считывая-записывая информацию по пути направления. Направление изменяется, если далее в данном направлении запросов нет.
- Пошаговое сканирование. Алгоритм аналогичен простому сканированию, но при движении учитываются только сектора существующих запросов. Сектора, запросы на которые поступили после начала движения в одном направлении, будут рассмотрены при движении в обратную сторону.
- Циклическое сканирование. Головка всегда движется только в одном направлении, а потом скачком возвращается. Обратим внимание на то, что если объединить пошаговое сканирование и циклическое сканирование, то будет исключена дискриминация крайних цилиндров.

7.5. Заключение

7.5.1. Выводы и рекомендации

Новости рынка программного обеспечения:
вышла русская версия Windows XP
в переводе Гоблина.
Программистский фольклор

Многие программисты знакомы с операционными системами в основном в меру тех помех, которые они создают при работе. Приносимая ими польза, как говорится, незаметна по определению.

Настоящее время может быть охарактеризовано как время доминирования двух семейств операционных систем: Windows и Unix. Каждое из них прочно заняло свою "нишу" и аппаратную платформу и стало там очень популярным. Впрочем, это не останавливает клонов и представителей семейств от постоянного и довольно успешного позиционирования на "чужой территории". Вот очень краткая сравнительная характеристика двух этих семейств.

- Семейство операционных систем Windows:
  - распространено, в основном, на персональных компьютерах, но стремится занять рынок серверных операционных систем;
  - системы, как правило, используются в качестве клиентских операционных систем в сетях;
  - системы имеют удобный графический интерфейс;
  - одна из основных целей — создание окружения, удобного для неопытных пользователей;
  - используются пользователями в широком диапазоне: от "чайников" до профессионалов;
  - собственно версии Windows проектировались большим количеством людей. Первые версии базировались на основателе семейства — операционной системе MS-DOS;
  - системы далеко не всегда надежны.
- Семейство операционных систем Unix:
  - распространено на мощных серверах и рабочих станциях, но с появлением клона Linux активно продвигается на рынок операционных систем для персональных компьютеров;
  - системы, как правило, используются в качестве серверных операционных систем в сетях;
  - исторически долгое время использовались только с интерфейсом командной строки;
  - одна из основных целей — создание окружения, удобного для развития программ;
  - используются широким кругом пользователей, в основном профессионалами;
  - первая версия была спроектирована и построена двумя исключительно талантливыми людьми — Кеном Томпсоном и Деннисом Ритчи;
  - достаточно надежны.

О надежности Solaris

О надежности конкретных диалектов Unix известно много фактов. Например, Solaris интенсивно использовалась в одном из университетов без единой перезагрузки в течение четырех лет. Когда на рабочую станцию пришла пора ставить обновленную версию операционной системы, ее физическое местоположение в серверной комнате долго не могли найти, поскольку в этом четыре года не было необходимости.

В большинстве случаев выбор операционной системы для реализации большинства обычных приложений вырождается в выбор между семействами Windows и Unix. Более интересен выбор операционных систем реального времени, о котором поговорим более подробно. На сегодняшний день на рынке представлено около двух сотен различных операционных систем реального времени. Выбор начинается с анализа архитектуры операционной системы и ее основных возможностей. Следующий шаг — оценка кода, являющегося связующим звеном между системой и аппаратурой (board support package), оценка аппаратной поддержки и драйверов. Если эта часть системы обладает недостатками, то риск неудачи проекта становится достаточно высоким. Завершающим этапом является количественная оценка временны*х* характеристик системы. Выход за пределы временны*х* ограничений может повлиять на выбор аппаратной части, пересмотр требований приложения и, в итоге, на отказ от использования выбранной операционной системы реального времени и переходу к рассмотрению следующей.

7.5.2. Проблемы и перспективы развития

Сообщение перед выключением компьютера: Вашу жену зовут Ольга, детей — Катя и Саша. Вы все еще хотите выйти из Windows?

Программистский фольклор

Независимые аналитические компании, например Gartner Group (**http://www3.gartner.com/Init**) и IDC (**http://www.idc.com**), ведут постоянный анализ тенденций в области развития операционных систем. Одной из основных тенденций является возрастание роли операционной системы Linux в качестве системы и для серверов, и для персональных компьютеров. Linux все чаще выбирают как отдельные программисты, так и целые компьютерные компании, и даже государства. Одновременно будет происходить замещение старых версий Windows новыми, но не столь быстрыми темпами как это происходило ранее.

Большинство пользователей интересует не сама операционная система, а наличие популярных (коробочных или скачиваемых из сети Интернет) приложений под операционными системами, в том числе офисных и бухгалтерских приложений. В противовес хорошо известному пакету MS Office (компании Microsoft (**http://www.microsoft.com**)) в операционной системе Windows, был разработан и получил огромную популярность аналогичный ему по возможностям пакет в Unix. Более того, распространяется два

пакета: коммерческий StarOffice (компании Sun Microsystems, Inc. (**http://wwws.sun.com/software/star/staroffice/**)) и свободно распространяемый (при поддержке той же Sun Microsystems) пакет OpenOffice (**http://www.openoffice.org**).

Выводы, касающиеся будущего, можно делать и анализируя динамику рынка операционных систем в недалеком прошлом. В табл. 7.2 приводится оценка изменения количества инсталляций для ряда операционных систем в период с 1996 по 2000 год. Рост количества инсталляций как новых версий систем из семейства Windows, так и систем Unix легко предсказать на основе данных в таблице.

Таблица 7.2. *Инсталляции различных операционных систем*

Операционная система	Основной микропроцессор	Другие микропроцессоры	Инсталляции (1996 год)	Инсталляции (2000 год)
MS-DOS	x86	Pentium	180 000 000	5 000 000
Windows 3.*	x86	Pentium	130 000 000	5 000 000
Windows 9x, 2K	Pentium	x86	60 000 000	300 000 000
Windows NT	Pentium	Power PC, Alpha	3 000 000	40 000 000
IBM OS/2	Pentium	x86	12 000 000	25 000 000
Macintosh	Power PC	680	25 000 000	35 000 000
Sun Solaris	SPARC	Pentium	1 500 000	10 000 000
Unix	Много	Много	6 000 000	20 000 000

Обратим внимание на поддержку процессоров двумя основными семействами операционных системам (табл. 7.3). Семейство Unix, в отличие от своего основного конкурента, поддерживает все современные известные процессоры.

Таблица 7.3. *Поддержка микропроцессоров операционными системами*

Семейство операционных систем	Pentium (Intel)	SuperSparc (Sun)	PA7100 (HP)
Windows	Windows	—	—
Unix	Linux, Solaris	Solaris, SunOS	HP-UX

Во всех операционных системах (особенно семейства Windows) есть множество дыр, которые хорошо известны. Разработчики пытаются их латать, но новые бреши отыскиваются постоянно. В связи с этим большой интерес вызывает компьютерная система Секьюрити, созданная под руководством члена-корреспондента РАН, профессора Бориса Арташесовича Бабаяна (**http://www.mcst.ru/**). Идеология системы, в которой от рождения существует иммунитет против любых вирусов, была заложена еще в первых отечественных суперкомпьютерах Эльбрус.

Наконец, подчеркнем, что развитие операционных систем подстегивают новые требования и новые достижения в области аппаратуры. Операционные системы должны эффективно использовать 64-разрядное адресное пространство современных микропроцессоров. Должны появиться распределенные операционные системы промышленного уровня. Операционные системы для встроенных систем — это большое направление, в котором есть простор для разработчиков.

7.5.3. Рекомендации по литературе

Искусство чтения состоит в том, чтобы знать, что пропустить.
Филип Хамертон

Всю литературу по операционным системам можно разделить на две группы. Первая содержит классическую теорию операционных систем.

- "Операционные системы" [Столлингс 2002]. Эта книга представляет собой полное и систематизированное введение в теорию операционных систем.
- "Современные операционные системы" [Таненбаум 2002а]. Всемирный бестселлер включает в себя сведения о последних достижениях в области технологий операционных систем.
- "Компьютерные сети" [Таненбаум 2002б]. В книге подробно и последовательно изложены основные концепции, определяющие современное состояние и тенденции развития компьютерных сетей.
- "Введение в операционные системы" [Иртегов 2002]. В учебнике рассматриваются архитектуры и принципы построения современных операционных систем.
- "Distributed operating system" [Tanenbaum 1995]. Книга является одним из лучших учебников по распределенным операционным системам.

Вторая группа содержит книги по конкретным операционным системам.

- По операционной системе Windows:
  - "Внутреннее устройство Microsoft Windows 2000" [Соломон, Руссинович 2001];
  - "Персональные компьютеры в сетях TCP/IP" [Хант 1997].

- По операционной системе Unix:
  - "Операционная система UNIX" [Робачевский 1997];
  - "UNIX: руководство системного администратора" [Немет, Снайдер, Сибасс, Хейн 1999];
  - "A Quarter Century of UNIX" [Salus 1994].

Содержание данной главы соответствует следующим международным и отечественным рекомендациям.

- "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Операционные системы". Основные курсы этой темы:
  - обзор операционных систем;
  - основы операционных систем;
  - параллелизм;
  - планирование и диспетчеризация;
  - управление памятью;
  - управление устройствами;
  - безопасность и защита данных;
  - файловые системы;
  - встроенные системы и системы реального времени;
  - отказоустойчивость;
  - оценка производительности системы.
- Специальность 351500 "Математическое обеспечение и администрирование информационных систем" имеет дисциплину специализации "Операционные системы и оболочки".

7.5.4. Вопросы и задания

1. Каковы основные функции операционной системы?
2. В чем заключались главные цели разработчиков первых операционных систем?
3. Как можно классифицировать операционные системы в зависимости от степени централизации?
4. Какие функциональные возможности следует добавить к централизованной операционной системе, чтобы она стала сетевой?
5. Что такое ядро операционной системы?
6. Перечислите классы операционных систем в зависимости от типа архитектуры ядра системы.

7. Опишите, какими этапами отличается выполнение системного вызова в микроядерной и слоистой операционных системах.
8. Укажите доминирующие в настоящее время операционные системы и объясните причины их популярности.
9. В чем состоят современные тенденции развития операционных систем?
10. Какие действия выполняет операционная система при создании нового процесса?
11. С какой аппаратной концепцией тесно связано понятие сигналов?
12. В чем состоит принципиальное отличие двух состояний процесса: ожидания и готовности?
13. Каковы основные подходы к решению задачи синхронизации процессов в централизованных архитектурах?
14. Назовите как минимум три задачи, для решения которых удобно применять семафоры.
15. В чем различие между бинарными и общими семафорами?
16. Приведите пример простого тупика с участием двух процессов и двух ресурсов и начертите соответствующий граф распределения ресурсов.
17. Сформулируйте четыре необходимых условия существования тупика.
18. В чем заключается аппаратная поддержка взаимоисключений?
19. Каковы основные подходы к синхронизации процессов в распределенных системах?
20. Каким образом должен выглядеть алгоритм планирования, чтобы он учитывал и интерактивные, и большие вычислительные задачи.
21. Можно ли задачу планирования процессов целиком возложить на приложения?
22. Какие события вызывают перепланирование процессов?
23. Назовите несколько причин использования многоуровневых протоколов.
24. В каких случаях можно использовать ненадежные примитивы передачи сообщений?
25. В чем состоит основное назначение механизма удаленного вызова процедуры?
26. Каковы принципы адресации в сетях TCP/IP?
27. Каковы принципы маршрутизации в сетях TCP/IP?
28. Как выглядит типичный процесс формирования сети?
29. В чем разница между физическим и виртуальным адресами?
30. Каковы преимущества и недостатки большого размера страницы?
31. В чем разница между внутренней и внешней фрагментациями?

32. Укажите различные методы, применяемые для преобразования виртуальных адресов в реальные при страничной организации памяти.
33. Какую основную задачу решает буфер быстрого преобразования адреса (TLB)?
34. С какой целью во многих файловых системах характеристики файла отделяются от его имени?
35. Что такое файловая система?
36. Каким из двух типов драйверов (блок-ориентированным или байт-ориентированным) обычно обслуживается диск?
37. Каковы основные причины для иерархического построения файловых систем?
38. Перечислите проблемы, возникающие при реализации распределенных файловых систем.
39. В условиях очень малых нагрузок практически все дисциплины планирования работы дисковой памяти вырождаются в одну схему. Какую?
40. Назовите несколько особенностей обычной операционной системы, которые не нужны во встроенной системе.
41. Какова цель проектирования и построения отказоустойчивых операционных систем?
42. Каковы методы достижения согласия в отказоустойчивых системах?
43. В чем основное отличие операционных систем реального времени?
44. Перечислите основные обязанности системного администратора.
45. Какие основные стратегии применяются для резервирования и восстановления файловых систем?
46. Назовите причины появления оболочек с развитым графическим пользовательским интерфейсом и факторы, влияющие на их развитие.
47. Укажите некоторые проблемы, связанные с безопасностью работы в сети Интернет.
48. Зачем в некоторых операционных системах проводится различие между стандартным выводом и стандартным потоком ошибок, если по умолчанию им соответствует одно и то же устройство — терминал?

ЛИТЕРАТУРА

Не существует ответов, есть лишь ссылки.
Библиотечный закон Винера

1. *Алексеев, Захарова, Русаков 2000* — Алексеев И. В., Захарова М. Н., Русаков А. И. Технологии IPv6 — Интернет нового поколения в мире и России // Информационное общество, № 2, 2000.

2. *Баурн 1986* — Баурн С. Операционная система Unix. — М.: Мир, 1986.

3. *Блэк 1990* — Блэк Ю. Сети ЭВМ: протоколы, стандарты, интерфейсы. — М.: Мир, 1990.

4. *Дейт 2000* — Дейт К. Дж. Введение в системы баз данных (6-е изд.). — К.; М.; СПб: Издательский дом "Вильямс", 2000.

5. *Дейтел 1987* — Дейтел Г. Введение в операционные системы. — М.: Мир, 1987.

6. *Дунаев 1995* — Дунаев С. Операционная система UNIX System V Release 4.2. — М.: Изд-во "МИФИ — Диалог", 1995.

7. *Дунаев 1998* — Дунаев С. UNIX сервер. Том 1, 2. — М.: Изд-во "МИФИ — Диалог", 1998.

8. *Зегжда, Ивашко 1997* — Зегжда Д. П., Ивашко А. М. Как построить защищенную информационную систему. — СПб.: Мир и семья-95, 1997.

9. *Иртегов 2002* — Иртегов Д. В. Введение в операционные системы. — СПб.: БХВ-Петербург, 2002.

10. *Кейлингерт 1985* — Кейлингерт П. Элементы операционных систем. — М.: Мир, 1985.

11. *Кейслер 1986* — Кейслер С. Проектирование операционных систем для малых ЭВМ. — М.: Мир, 1986.

12. *Колин 1975* — Колин А. Введение в операционные системы. — М.: Мир, 1975.

13. *Корнеев 1999* — Корнеев В. В. Параллельные вычислительные системы. — М.: "Нолидж", 1999.

14. *Кулаков, Луцкий 1998* — Кулаков Ю. А., Луцкий Г. М. Компьютерные сети. — Киев: Юниор, 1998.

15. *Немет, Снайдер, Сибасс, Хейн 1999* — Немет Э., Снайдер Г., Сибасс С., Хейн Т. UNIX: руководство системного администратора. — СПб.: BHV — Санкт-Петербург, 1999.

16. *Николаев 1997* — Николаев Ю. И. Проектирование защищенных информационных технологий. — СПб.: Изд-во СПбГТУ, 1997.

17. *Олифер, Олифер 2001* — Олифер В. Г., Олифер Н. А. Сетевые операционные системы. — СПб.: Питер, 2001.

18. *Робачевский 1997* — Робачевский А. М. Операционная система UNIX. — СПб.: BHV — Санкт-Петербург, 1997.

19. *Соломон, Руссинович 2001* — Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. — СПб.: Питер, М.: Издательско-торговый дом "Русская Редакция", 2001.

20. *Таненбаум 2002а* — Таненбаум Э. Современные операционные системы. — СПб.: Питер, 2002.

21. *Таненбаум 2002б* — Таненбаум Э. Компьютерные сети. — СПб.: Питер, 2002.

22. *Хант 1997* — Хант К. Персональные компьютеры в сетях TCP/IP. — К.: Издательская группа BHV, 1997.

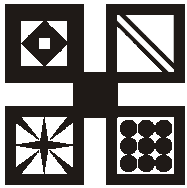
23. *Lewine 1995* — Donald A. Lewine. POSIX Programmer's Guide. O'Reilly & Associates, Inc. 1995.

24. *Lewis, Berg 1995* — Bil Lewis, Daniel J. Berg. A Guide to Multithreaded Programming. Threads Primer. SunSoft Press. A Prentice Hall Title. 1995.

25. *Salus 1994* — Peter H. Salus. A Quarter Century of UNIX. Addison-Wesley Publishing. 1994.

26. *Stevens 1994* — W. Richard Stevens. TCP/IP Illustrated: the protocols. Addison-Wesley Publishing. 1994.

27. *Tanenbaum 1995* — Andrew S. Tanenbaum. Distributed operating system. Prentice-Hall, Inc. 1995.



Глава 8

Элементы менеджмента и социальные вопросы информатики

В этой главе мы рассмотрим несколько вопросов, объединенных понятием "человеческий фактор".

- Как сделать карьеру в области информационных систем? В чем особенности национальной карьеры?
- Какими должны быть первые шаги в поиске работы? Как правильно составить резюме? Как проводится собеседование и как к нему подготовиться?
- Почему некоторые программисты на порядок полезнее других?
- Что такое олимпиадное (соревновательное) программирование? Какая от него польза и связано ли оно с реальным программированием?
- Каковы этические проблемы, возникающие при разработке программного обеспечения? Как их можно решать с этической и технической точек зрения?

8.1. Введение в элементы менеджмента и социальные вопросы информатики

Кто хочет обрести счастье или мудрость,
тот должен искать перемен.
Конфуций

Профессиональные технические вопросы должны быть основными в багаже знаний программиста. Однако ими нельзя ограничиваться. Слишком многие первоклассные программисты с пренебрежением относятся к знаниям в других областях или считают, что уровень компетенции в своей области заменит им знание других предметов. К сожалению, очень часто причиной неудовлетворительных результатов работы становится недостаток или

пренебрежение знаниями, не относящимися к прямой компетенции специалиста. Существуют еще два важных направления, которые полезно знать дополнительно или, по крайней мере, понимать точки соприкосновения с ними.

Первое направление — это управление (менеджмент), о котором шла речь в *разд. 3.2.2*. Так или иначе, работа программистов постоянно связана с работой менеджеров. Как правило, это менеджеры программных проектов, являющиеся управляющими первичного звена. Рассмотрим основные функциональные области менеджмента.

- *Стратегический менеджмент* — это система управления основными аспектами деятельности на базе планирования и реализации долгосрочной концепции развития, определяющей общие цели фирмы, приоритеты в распределении ресурсов и инвестиционных вложений, конкурентные преимущества продукции компании на рынке. Стратегический менеджмент — это задача высшего менеджмента компании, как отмечалось в *разд. 3.2.2.1*.
- *Управление проектами* — это деятельность, направленная на реализацию проекта с максимально возможной эффективностью при заданных ограничениях по времени, денежным средствам и ресурсам, а также качеству конечных результатов проекта. Большая часть *разд. 3.2.2* посвящена этой теме.
- *Управление персоналом* — это система управления сотрудниками внутри организации, направленная на формирование кадрового потенциала и достижения необходимого производственного поведения, путем привлечения, мотивации и организации работы персонала. К этой категории относятся и вопросы набора новых сотрудников *(см. разд. 8.2)* и вопросы возможных путей развития карьеры программиста *(см. разд. 8.3)*, которые подробно обсуждаются далее.
- *Финансовый менеджмент* — это комплекс средств, инструментов и мероприятий, обеспечивающих принятие и реализацию финансовых решений. Это очень интересные вопросы, к сожалению, полностью выходящие за рамки нашей книги.
- *Маркетинг* — комплекс средств, инструментов и мероприятий, обеспечивающих максимальную ориентацию производственной, коммерческой и сбытовой деятельности компании на удовлетворение запросов заказчика. Основной инструмент маркетинга — *реклама*, которая представляет собой систему мер целенаправленного воздействия на потребителей, формирующих и регулирующих движение товаров на рынке. Эта область менеджмента подробно и интересно рассмотрена в [Викентьев 1998].
- *Связь с общественностью* — это инструмент менеджмента, основная стратегия которого — создание атмосферы доверия. Обратим внимание

на то, что его аудитория не имеет четко выраженных границ и адресов. Хорошими способами создания положительного общественного мнения являются благотворительность и меценатство. Эта область менеджмента также рассмотрена в [Викентьев 1998].

- *Менеджмент качества* — это совокупность элементов организационной структуры, методик, процессов и ресурсов, необходимых для создания, применения и поддержки общего руководства качеством программного продукта. О подходах к обеспечению качества программного обеспечения и оценке качества шла речь в *разд. 3.7*.

Второе направление связано с группой наук, которые обычно объединяют общим словом "социальные". Особо интересны следующие науки и учения, изучающие различные аспекты жизни человеческого общества.

- *Психология* — наука о закономерностях, механизме и фактах психической жизни человека. Различают фундаментальные отрасли психологии и прикладные (рис. 8.1), причем среди последних нас могут особенно заинтересовать две:
  - *психология программирования* исследует деятельность программиста, психологические закономерности приема и переработки информации, функционирование психических приемов памяти и мышления у программиста, его работоспособности, вопросы профессионального отбора. Многим программистам известна классическая работа [Weinberg 1998];
  - *психология управления* рассматривает вопросы создания и развития работающего коллектива, формирования в нем малых групп и выдвижения лидеров, психологические аспекты деятельности руководителя, психологии власти, исполнения и подчинения, осуществления контроля, а также кадровые вопросы. В качестве дополнительной литературы здесь предложим учебник [Психология 2000].

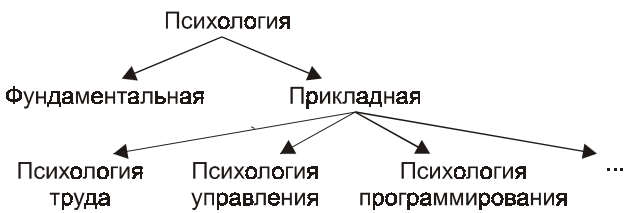


Рис. 8.1. Основные направления психологии

- *Педагогика* — наука о воспитании и обучении. Педагогика исследует сущность, цели, задачи и закономерности воспитания, образования и обучения, их роль в жизни общества и развитии личности. Подходы к обучению программированию кратко рассматривались в *разд. 1.3* и *1.4*.

- *Культурология* — это гуманитарная наука, изучающая закономерности развития и функционирования культуры, ее структуру и динамику, взаимосвязи и взаимодействия с другими сферами материальной и духовной жизни. Важность понимания программистом таких элементов, как культурная среда, была продемонстрирована в *разд. 4.7.3*.

- *Юридическая наука* — общественная наука, изучающая право как особую систему социальных норм и различные аспекты правоприменительной деятельности. Существует ряд так называемых компьютерных преступлений. К ним относятся взломы компьютерных систем, сетевые атаки, разработка вирусов, раскрытие конфиденциальной информации. Возможные правовые проблемы в области программного обеспечения и пути их решения рассматриваются в книге [Носова, Козадеров 1998].

- *Этика* — это учение о морали, ее сущности, структуре, функциях, законах, ее историческом развитии и роли в общественной жизни. Программисты должны понимать моральную ответственность, которая лежит на них, и возможные последствия неудачи. Небольшой экскурс в эти проблемы мы сделаем в *разд. 8.5*.

- *Философия* — форма общественного сознания, система взглядов на мир (мировоззрение) и на место человека в нем. "Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" утверждают, что программисту желательно разбираться в некоторых философских течениях, например, утилитаризме (в котором основой нравственности и критерием человеческих поступков является польза, понимаемая как средство достижения "наибольшего счастья для наибольшего числа людей") и деонтологии (теории проблем долга и моральных требований).

Приобретение знаний в перечисленных областях позволит полнее раскрыться всем Вашим способностям. Данная глава ориентирована на реальную помощь программистам в построении успешной будущей профессиональной деятельности.

8.2. Набор новых сотрудников

Этот раздел в одинаковой степени полезен и программистам, желающим устроиться на работу, и руководителям проектов, которые их на работу принимают. Начинающим программистам мы изложим требования, которые они могут встретить от принимающей стороны, не забыв и о требованиях, которые они должны предъявить сами себе.

8.2.1. Привлечение кандидатов

Купившему клетку для хомяка — хомяк бесплатно.

Объявление в зоомагазине

Набор новых сотрудников начинается с того, что компания должна сформулировать преимущества, которые может им предложить. Как правило, это более высокая заработная плата по сравнению с той, что кандидат имел раньше, или с той, что может предоставить конкурент. Также это могут быть более привлекательные условия, время и место работы, дополнительные социальные услуги и т. п. Известно, что службы многих компаний тщательно изучают деятельность компаний-конкурентов в использовании "человеческих ресурсов".

Набор кадров — это совокупность мер, предпринимаемых компанией для привлечения кандидатов на работу. Существует множество методов набора кадров [Цветаев 1999].

- Через инициативные обращения в компанию людей, желающих получить работу.
- Через друзей и родственников.
- Через сотрудников фирмы.
- Через объявления в специальных изданиях и службах массовой информации (например, в агентствах по найму в Интернете — **http://www.hro.ru** и **http://www.joblist.ru/**).
- Через частные и муниципальные агентства по трудоустройству.
- Через учебные заведения.
- Через использование базы данных кандидатур из предыдущего цикла набора.

Согласно статистике, наиболее эффективными методами поиска работы являются обращение непосредственно к предпринимателю и поиск через друзей и родственников.

Очень хорошо, если кандидат имеет положительную рекомендацию от одного (лучше от нескольких) из сотрудников, уже работающих в данной компании. Рекомендованный должен быть положительно охарактеризован как человек (в плане отношения к работе и коллегам) и как специалист в требуемой области. Желательно, чтобы рекомендующий имел личный опыт работы в одном коллективе с кандидатом. Заметим, что наличие положительной рекомендации само по себе ничего не гарантирует, а лишь позволяет отсеять случайных людей. Бывает полезно получить также отзывы о кандидате от незаинтересованной стороны (общие знакомые, бывшие коллеги рекомендуемого).

Общий совет поступающим на работу: верьте обещаниям нанимателя не больше, чем любой рекламе.

8.2.2. Анализ краткой биографии (резюме)

8.2.2.1. Основные советы по составлению резюме

Серьезная организация возьмет на высокооплачиваемую должность опытного хакера. Резюме просим размещать на главной странице сайта **www.microsoft.com**.

Анекдот о необходимости резюме при представлении себя компании

Специалисту нужно представить себя компании, а компании — получить информацию о специалисте. Для этого традиционно используется основной документ — резюме.

Краткая биография (резюме) — это изложение сведений об основных анкетных данных и профессиональных навыках. Включая необходимую и достаточную информацию об авторе, резюме является визитной карточкой при поиске работы. Резюме рассылается или вручается лично работодателю или кадровому агентству. Цель резюме — подчеркнуть все ваши положительные моменты и показать, чем вы лучше других.

Существует несколько основных советов по составлению резюме. Резюме должно быть:

- правдивым;
- избирательным (информация отбирается исходя из цели резюме);
- достаточно кратким (как правило, оно должно уместиться в пределах двух листов формата A4);

  О краткости

  Наша рекомендация о краткости верна не во всех случаях. Например, научные резюме характерны основательным перечислением достижений подателя резюме и его основных печатных работ.

- оформлено в строгом стиле;
- грамотно написано.

8.2.2.2. Структура резюме

Резюме для специалиста в области информационных технологий имеет некоторые отличия от классического варианта. Укажем сначала рекомендуемую структуру резюме.

1. Фамилия, имя, отчество.
2. Контактная информация (адрес, телефон, электронная почта).
3. Цель в поиске работы. Если вы претендуете на конкретную вакансию, то следует указать именно ее. Можно указать, на каких условиях вы хотите получить данную работу.
4. Текущая должность и место работы.5

5. Краткая профессиональная характеристика. Здесь следует указать те области специализации, в которых вы наиболее сильны.
6. Образование. Указывается с датами, в обратном порядке.
7. Профессиональные достижения. Здесь вы должны выделить профессиональную деятельность, которая дает вам основания претендовать на вакансию.
8. Профессиональный опыт. В обратном хронологическом порядке следует указать места работы, включая года работы, название предприятия и должность.
9. Знание платформ и инструментов, включая аппаратные, операционные и языковые платформы.
10. Иностранные языки. Следует указать степень владения языками (свободно или со словарем) и уровень (разговорный или технический).
11. Рекомендации. Как правило, рекомендации не прикладываются к резюме, а предоставляются по запросу, если он будет сделан.

Возможно, после первоначального варианта особо заинтересовавших кандидатов следует попросить расширить и дополнить резюме. Однако обратим внимание на то, что из первоначального варианта часто можно сделать некоторые интересные выводы.

8.2.2.3. Образец резюме

Я бесконечно признательна, что здесь, в Швеции,
мне дали возможность читать лекции,
несмотря на то, что я женщина

Софья Васильевна Ковалевская

Приведем образец составленного резюме.

Ковалевская Софья Васильевна

Адрес: 196240, Россия, Санкт-Петербург,
проспект Чебышева, д. 111, кв. 111

Электронная почта: sofia_kovalevskava@sparc.spb.su

Телефоны: +7(812) 999-1111 (рабочий)
+7(812) 999-2222 (домашний)
+7(812) 999-3333 (мобильный)

Цель

Получение должности руководителя отдела разработки компиляторов в компании "Компилятор++".

Текущая должность и место работы

Руководитель проектов по разработке и сопровождению компиляторов в Санкт-Петербургском Центре компьютерных технологий. Текущие обязанности включают:

- техническое руководство проектами по сопровождению компиляторов с языков программирования C, C++, Pascal;
- проектирование и разработку фрагментов программных продуктов;
- сопровождение языково-зависимой части компиляторов, включая исправление ошибок в программах, разработку тестов и сопровождение тестовой базы.

Краткая профессиональная характеристика

Специалист в области разработки компиляторов. В течение 15 лет участвовала в ряде крупных проектов в области создания компиляторов, инструментальных оболочек, экспертных систем, вычислительных программ.

Области научных исследований и профессионального интереса: методология сопровождения компиляторов, принципы инженерии программирования, параллельное программирование. Автор 12 научных публикаций.

Образование

1983 сентябрь — 1988 июнь

Санкт-Петербургский государственный университет, математико-механический факультет, кафедра информатики. Диплом специалиста с высшим профессиональным образованием. Специальность — математик, специализация — прикладная математика.

Профессиональные достижения

- Руководство компиляторными проектами. Руководила выпуском пяти версий программного продукта "Профессиональная среда разработчика программ на языке программирования C++".
- Разработка компиляторов. Реализована подсистема автоматического распараллеливания программного кода на основе механизма прагм согласно стандарту OpenMP. Выполнена реализация безопасных многопоточных библиотек, интернационализация компиляторов и библиотек. Разработана подсистема интерактивных расширенных подсказок пользователя по возникающим при компиляции синтаксическим ошибкам.
- Сопровождение компиляторов. Исправлено более полутора тысяч зарегистрированных ошибок в программных продуктах. Решена "проблема 2000 года" для семейства компиляторов.
- Аналитическая работа. Выполнена оценка и сравнение производительности компиляторов. Сделан сравнительный анализ расширений стандарта для компиляторов конкурирующих компаний.

- Развитие и сопровождение тестовых баз. Разработано более двух тысяч тестов для тестирования языков программирования C, C++, Pascal. Выполнено исследование тестового покрытия и покрытия документации для семейства компиляторов.
- Подготовка документации. Написано "Руководство пользователя по профессиональной среде разработчика программ на языке программирования C++".

Профессиональный опыт

1998 апрель — по настоящее время.

- Должность: Руководитель проектов по разработке и сопровождению компиляторов.
- Место работы: Санкт-Петербургский Центр компьютерных технологий, Санкт-Петербург.

1993 январь — 1998 март

- Должность: Инженер по разработке и сопровождению компиляторов.
- Место работы: Санкт-Петербургский Центр компьютерных технологий, Санкт-Петербург.

1988 июль — 1992 декабрь

- Должность: Старший научный сотрудник.
- Место работы: Институт технологии программирования при Санкт-Петербургском государственном университете, Санкт-Петербург.

Знание платформ и инструментов

Аппаратные платформы: Intel, SPARC.

Операционные системы: Windows (9x, 2000), Unix (Sun Solaris 2.*).

Языки программирования: C, C++, Java, Pascal, Prolog, Ada, Perl.

Системы управления базами данных: Oracle, MS Access, Sybase.

Инструментарий Интернета: CGI, HTML, XML, JavaScript.

Иностранные языки

Английский разговорный и технический свободно.

Рекомендации

Будут представлены по запросу.

8.2.2.4. Как не следует писать резюме

Русские называют дорогой то место,
где собираются проехать.

Зарубежный фольклор

Примеры того, как не надо писать резюме, взяты с сохранением авторской орфографии из различных конференций FIDO.

- "... Из языков программирования хорошо знаю ASM, Visual C, самостоятельно разрабатывал базы данных на Асме, как под DOS, так и под Windows. Кроме баз данных на Асме пишу разнонаправленные программы (начиная с драйверов, заканчивая полнофункциональными приложениями), в общем я просто фанат ассемблера, но также сейчас увлекаюсь Visual C, хотя раньше писал на Borland C. Могу оптимизировать алгоритмы, если не устраивает их медленная работа ..."
- "... Воплощен с 1967 г. от Р. Х. Детсад, школа (был командиром октябрятской звездочки), техникум, служба в СА (РВСН), институт ЛИИЖТ (выч. техника), женился-развелся ..."
- "... Несмотря на то, что мне 16 лет я могу: настроить сеть, собрать разобрать комп, знаю Pascal, c++, прикладные программы, могу работать в ночное время, на большую зарплату не притендую ..."
- "... Готов работать 24 часа в сутки ..."
- "... На диплом писал программу на C по проектированию носочной части обувной колодки. Обучаясь на военной кафедре, получил специальность Начальник вещевой службы (отдела снабжения) ..."
- "... Чего умею: Сетки (администрить, прокладывать, проектировать — короче создавать), програмить (знаю c++, но давно ничего не писал), работать (офисы, корелы, фотошопы и т. п.), собирать и настраивать компы, водить машину (стаж 4 года), знаю Интернет технологию, быстрообучаем ..."
- "... Я понимаю, что моя ценность как молодого специалиста невелика, но готов работать и за относительно небольшую зарплату ..."
- "... Легко обучаема, исполнительна, коммуникабельна, порядочна, приятная внешность ..."
- "... В совершенстве владею персональным компьютером, его аппаратной частью и рядом программ ..."
- "... Рублю в комп. железе, как и полагается. Ищу работу где нужен интеллект! ..."
- "... Разговорный английским в степени, достаточной для изъяснения ..."
- "... Прошу рассмотреть мою кандидатуру на работу в зоне Internet ..."

- "... Я была бы рада получить приглашение на собеседование, чтобы при личной встрече обосновать свои возможности приносить пользу вашей фирме ..."
- "... Немного программист ..."
- "... Склонность к глубокому изучению предмета в деталях, вплоть до самого нижнего уровня ..."
- "... Speak English fluently, computer — literate, major — English language and literature, drive — license, married, son of 7, goal — oriented, hard — working, ready work under pressure ..."
- "... поиск новых технологий программирования производства светильников ..."

Общий совет нанимателю: верьте резюме поступающего на работу не больше, чем любой рекламе.

И еще. Первый пример написан очень скверно, но искренне. Следует рискнуть, как минимум, пригласить такого кандидата на собеседование, чтобы решить, стоит ли предложить ему пройти испытательный срок.

8.2.3. Собеседование

8.2.3.1. Общие рекомендации

Жалеть о сказанном мне приходилось,
о не сказанном — никогда.

Ксенократ

Собеседование является основным методом оценки и отбора кандидатов при приеме на работу.

Собеседование проводит комиссия, состоящая из:

- руководителя подразделения, в которое отбирается кандидат;
- нескольких сотрудников подразделения, в которое отбирается кандидат;
- руководителей и сотрудников незаинтересованного подразделения.

> ***О собеседовании с психологом***
>
> Многие компании сейчас приглашают на собеседование психолога. В таком случае часть собеседования проходит отдельно, один на один с психологом. Естественно, в первую очередь психолог обращает внимание на личностные качества кандидата.

Дадим несколько рекомендаций как кандидатам, так и комиссии, проводящей собеседование.

- Обычно собеседование с одним кандидатом занимает не меньше часа. Кандидатов необходимо уведомить о встрече заблаговременно.

- Кандидатов следует заранее попросить принести для показа свои наиболее удачные программные разработки. Это должны быть фрагменты исходных текстов программ для демонстрации стиля программирования.
- В ходе интервью необходимо оценить ряд качеств кандидатов. Выделяют четыре группы характеристик, причем вторую, третью и четвертую следует попытаться оценить количественно (например, по пятибалльной шкале) [Цветаев 1999].
  - *Формальные качества* (подтверждаемые документами) — возраст, семейное положение, гражданство, профессия, специальность в рамках профессии.
  - *Профессиональные качества* — уровень квалификации в области программирования, профессиональная динамика, опыт работы, тип акцентуации в трудовом поведении (пунктуальность, аккуратность, стремление довести дело до конца) и др.
  - *Личностные качества* — целеустремленность, преданность, открытость, дружелюбие, контактность и т. д.
  - *Интеллектуальные качества* — уровень развития, способность к обучению и пр.
- Кандидат должен тщательно подготовиться к деловой встрече, собрав максимальное количество доступной информации о компании, в которую он идет на собеседование. Эта информация позволит получить представление о программных продуктах, производимых компанией, и целенаправленно готовиться к собеседованию.
- Кандидат должен произвести благоприятное впечатление на комиссию. Говорить он должен уверенно и четко. Полезные рекомендации по стилю поведения даются в книгах [Карнеги 1991], [Пиз 1992] и [Холопова, Лебедева 1995]. Внешний вид кандидата должен соответствовать стилю, принятому в компании. Большинство компьютерных компаний поддерживают свободный и демократичный стиль одежды (джинсы и футболка), однако ряд компаний требует от сотрудников строгую деловую одежду.

8.2.3.2. Вводная часть собеседования

Не дайте себя обмануть в другом месте —
покупайте у нас!

Объявление в магазине

Во вводной части собеседования интервьюеры должны сделать так, чтобы кандидат чувствовал себя максимально комфортно. Следует представить кандидату всех участвующих в собеседовании и рассказать о том, как собеседование будет проходить. Кандидат должен знать, что комиссия будет обращать внимание на то, как он подходит к решению проблемы. Разумно

проводить собеседование в месте, позволяющем избавиться от формальных барьеров (например, стола), разделяющих кандидата и комиссию.

8.2.3.3. Предыдущие проекты кандидата

Всякому теперь кажется, что он мог бы наделать добра на месте
и в должности другого и только не может сделать этого
в своей должности. Это причина всех зол.

Н. В. Гоголь

Если кандидат имеет опыт работы, то он должен сделать короткое сообщение о своей прошлой и текущей работе. При этом могут возникнуть дополнительные вопросы к кандидату.

- Почему он ищет работу?
- Почему он прислал свое резюме именно к нам?
- Что его не устраивает на нынешнем месте работы?
- Насколько быстро кандидат может оставить свое нынешнее место работы?

Из информации о предыдущих проектах необходимо выяснить, какой опыт реальной работы есть у кандидата. Следует оценить:

- участие в серьезных производственных проектах;
- опыт создания и сопровождения программного обеспечения в той области специализации, для работы в которой отбирается кандидат;
- опыт коллективных разработок и опыт руководства коллективом;
- опыт сопровождения программного обеспечения (работа с чужим кодом).

Как известно, предыдущее поведение является лучшим индикатором поведения будущего. Если для кандидата важными будут являться определенные требования, умения и качества, следует узнать, как ему удавалось применять их на предыдущих местах работы.

8.2.3.4. Вопросы на размышление

Мужчины любят женщин, женщины любят детей,
дети любят хомячков, хомячки никого не любят.

Эйлис Эллис

Идея вопросов на размышление заключается в том, что в подавляющем большинстве случаев кандидат не может знать ответ на них. У интервьюеров появляется хорошая возможность посмотреть на подход кандидата, как он возьмется за поиск ответа (**http://www.joelonsoftware.com/**). Приведем примеры таких вопросов.

- Каков вес Сфинкса на Университетской набережной в Санкт-Петербурге?

- Сколько автомобильных заправок в Санкт-Петербурге?
- Сколько фонтанов в пригороде Санкт-Петербурга — Петергофе?

Все кандидаты делятся на две группы по отношению к таким вопросам.

- *Смышленые кандидаты*, понимающие, что здесь проверяются не энциклопедические знания, а способность искать путь к правильному (или, по крайней мере, достаточно разумному) ответу. Одним из возможных вариантов начала рассуждения на второй вопрос может быть таким: "Население Санкт-Петербурга составляет около двух миллионов человек. Примерно четверть жителей имеют автомобили..." Если даже это и не совсем верно, важен энтузиазм, с которым кандидат пытается решить задачу. В конце концов, кандидат может просто попросить справочник "Санкт-Петербург. Желтые страницы". Любые разумные ответы уже являются хорошим знаком.
- *Несообразительные кандидаты*, впадающие в уныние и расстройство после получения таких вопросов. Можно предложить таким кандидатам некоторые подсказки и посмотреть, смогут ли они с их помощью начать двигаться в правильном направлении.

Об игре "Что? Где? Когда?"

Заметим, что более сложными, но близкими по идее к вопросам на размышление, являются вопросы на эрудицию и сообразительность игры "Что? Где? Когда?". Предложим читателю подумать, например, над таким вопросом: "Четверостишие из современного народного творчества: Писателям, временно севшим на мель, / Сегодня дорога открыта. / Роман напишите на HTML: / ... / В последней строке стоит измененное название литературного произведения. В нем к двум, по сути, именам собственным, добавлена одна и та же современная приставка. Воспроизведите эту строку".

Существуют шуточные вопросы, с помощью которых можно попробовать оценить умственные способности кандидатов.

- Вопрос 1. Как поместить жирафа в холодильник?
  - Правильный ответ. Открыть холодильник, засунуть туда жирафа, закрыть холодильник.
  - Комментарий к вопросу. Этот вопрос позволяет выяснить, нет ли у кандидата склонности искать чересчур сложные решения для простых задач.
- Вопрос 2. Как поместить в холодильник слона?
  - Неправильный ответ. Открыть холодильник, засунуть туда слона, закрыть холодильник.
  - Правильный ответ. Открыть холодильник, вынуть оттуда жирафа, засунуть туда слона, закрыть холодильник.

- Комментарий к вопросу. Этот вопрос позволяет выяснить, способен ли кандидат при принятии решений учитывать последствия своих предыдущих действий.

❒ Вопрос 3. Лев созвал всех зверей на собрание. Явились все, кроме одного. Что это за зверь?

- Правильный ответ. Это слон. Он же в холодильнике, помните?
- Комментарий к вопросу. Этот вопрос проверяет память кандидата.

❒ Вопрос 4. Вам нужно пересечь широкую реку, которая кишит крокодилами. Как вы это сделаете?

- Правильный ответ. Вплавь. Ведь крокодилы-то все на собрании у льва.
- Комментарий к вопросу. Этот вопрос позволяет выяснить, способен ли кандидат учиться на собственных ошибках.

8.2.3.5. Вопросы о проектировании архитектуры

Убийцы и архитекторы всегда возвращаются
на место преступления.

Питер Устинов

Кандидату может быть предложено что-либо спроектировать. Известно, что Джейб Блюменталь (Jabe Blumenthal) (автор первоначального проекта Excel) любил давать кандидатам задание спроектировать дом. Если кандидат после этого немедленно рисовал прямоугольник, то такой кандидат немедленно отвергался.

На что следует обращать внимание при ответе на такие вопросы? Хорошие кандидаты сразу попробуют получить как можно большее количество информации от комиссии. Более того, имеет смысл сразу отвергать кандидатов, не задавших вопрос "А для кого этот дом предназначается?".

Вот еще два интересных задания для проектирования, над которыми вам предлагается подумать самостоятельно:

❒ спроектировать мусорные урны, предназначенные для установки на уличных перекрестках;

❒ спроектировать набор баночек со специями, предназначенный для слепых людей.

8.2.3.6. Технические вопросы

Отплякиваясь от сурых пляк, каждый хамсик шмыряет на глын по 5 гнусиков.
Сколько гнусиков шмырнут на глын 12 хамсиков,
отплякивающихся от сурых пляк?

Г. Остер. "Задачник"

Технические вопросы позволяют получить информацию о технической компетентности кандидата. Для кандидата в разработчики системного про-

граммного обеспечения возможный список технических групп вопросов может быть следующим:

- знание операционных систем семейства Unix (опыт использования, программирования и администрирования);
- знание операционных систем семейства Windows (опыт использования, программирования и администрирования);
- знание локальных сетей (опыт использования, программирования и администрирования);
- знание сети Интернет;
- владение методологией и практическим опытом объектно-ориентированного программирования на языке C++;
- знание языка программирования C;
- знание Java (языка и окружения);
- знание элементов тестирования программных продуктов.

> ***О технических вопросах***
>
> Существует точка зрения (**http://www.joelonsoftware.com/**), согласно которой технические вопросы в процессе собеседования являются совершенно бессмысленными. Собственно, это было известно давно и отразилось в многочисленных фольклорных историях. Например, рассказывают что Эдисон долго не мог подобрать себе помощника, предлагая кандидатам вопросы типа "Из чего делают нержавеющую сталь?" Эйнштейн, когда Эдисон пожаловался ему на эту ситуацию, пробежав глазами вопросник, сказал: "Не дожидаясь отказа, свою кандидатуру снимаю сам". Не существует связи между знанием (или незнанием) кандидатом ответа на конкретный технический вопрос и тем, будет ли он принят на работу. Кстати, ответы на большинство технических вопросов при необходимости могут быть найдены в онлайновых справочниках за считанные секунды.

Примеры вопросов для оценки технической компетентности

Каждому кандидату во время интервью задается по 3—4 вопроса из каждой области, знания в которой требуется оценить. Часть технических вопросов может быть "провокационного" характера. Они позволят выяснить способность кандидата к нестандартным рассуждениям. Часть вопросов поможет определить общий объем интересов и знаний, а также понимание перспектив развития в указанной предметной области. Приведем некоторые возможные вопросы. Попробуйте потренироваться в ответах на них.

- Язык C.
  - Что такое глобальная статическая переменная?
  - Что такое локальная статическая переменная?
  - Для чего используется объединение (union)?

- От каких выражений нельзя брать адрес (&)?
- Что такое директивы условной компиляции?
- Где возможно использование битовых полей?

❒ Объектно-ориентированное программирование и C++.

- Что такое виртуальная функция? В чем ее отличия от обычной? Приведите примеры.
- Что такое инкапсуляция (наследование, полиморфизм)?
- Что такое абстрактная функция? Какой синтаксис ее описания в C++?
- Для чего используются шаблоны (templates)?
- Перечислите все способы использования ключевого слова "const" в C++.
- Что такое защищенное наследование (protected inheritance)?
- Чем отличаются указатели (pointers) и ссылки (references) в C++?
- В чем отличия класса (class) и структуры (struct)?
- Две одинаковые программы написаны на C и на C++. Какая из них работает быстрее?
- Докажите, что объектно-ориентированное программирование вредно и опасно.

❒ Язык программирования Java.

- Есть ли в Java множественное наследование?
- Как в Java реализована многонитевость (multithreading) и синхронизация нитей?
- Что такое финализация?
- Перечислите отличия интерфейса (interface) от абстрактного класса (abstract class)?
- Что такое сериализация?
- Есть ли в Java виртуальные функции? Если да, то какой синтаксис определения виртуальной функции?
- Для чего используется рефлексия?
- Назовите несколько классов из пакета java.io (java.lang, java.awt, java.net) и поясните их предназначение.
- Назовите основные отличия Java от C++.
- Java — это компилируемый или интерпретируемый язык? Расскажите все известные вам подробности.

- Операционные системы семейства Unix.
  - Какая команда выдает список процессов в системе?
  - Что такое каналы (pipes)? Как создать канал из командной строки?
  - Что такое сигналы (signals)? С помощью какой команды можно послать процессу сигнал?
  - Какую функцию выполняет системный вызов `kill()`?
  - В чем отличие символических ссылок (symbolic links) от обычных?
  - Для чего нужен X-сервер (X-server)?
  - Как восстановить удаленный файл?
  - С помощью какого системного вызова создаются процессы?
- Интернет и сетевые технологии.
  - Назовите несколько протоколов, проколов и приколов, широко используемых в Интернете.
  - Что такое DNS?
  - Что такое CGI-интерфейс?
  - Что такое апплет (applet)?
- Тестирование.
  - Что такое тестирование методом черного (белого) ящика?
  - Что такое тестовая база?
  - Как был организован процесс тестирования на Вашем предыдущем месте работы?

Технические вопросы в тестах

В случае необходимости технического собеседования с большим числом кандидатов можно предложить им выполнить письменное тестирование. Однако это следует делать только в исключительных случаях, например, когда количество кандидатов очень велико, а количество времени на собеседование очень мало. Результаты, которые получаются в итоге такого тестирования, имеют довольно слабое отношение к реальным знаниям кандидата. Самое плохое, что они не показывают, как кандидат умеет мыслить и излагать свои идеи.

Предложим вариант тестов, состоящий из 10 вопросов (часть из них взята из работы [Clifton 1995]).

1. Рассмотрим фрагмент программы:

```
...
    i = 0;
    j = 0;
```

```
    for ( j = 1; j < 10; j++ ) {
        i = i + 1;
    }
...
```

Каким будет значение переменной `i` после того, как код будет выполнен?

(А) 0;

(Б) 1;

(В) 3;

(Г) 9;

(Д) 10.

2. Иногда объектный модуль, создаваемый компилятором, включает информацию (символьные таблицы), представляющую карту соответствия имен в программе их адресам. Наиболее вероятная цель этой информации:

(А) для использования в качестве входной информации отладчика;

(Б) для повышения эффективности выполнения программы;

(В) для сокращения размера символьных таблиц, используемых компилятором;

(Г) для указания загрузчику, где находится каждая переменная;

(Д) в качестве вспомогательной информации операционной системе.

3. Большинство руководств по языку программирования определяют семантику языка, используя неформальные описания и примеры. Что является причиной того, что формальные методы для описания семантики не имеют такого широкого распространения?

(А) Формальные методы описания семантики очень плохо изучены на данный момент.

(Б) Наиболее известные формальные методы описания семантики — операционные, денотационные и аксиоматические — непригодны для достаточно полного описания семантики языков программирования.

(В) Неформальные методы обычно полностью решают задачу описания языков программирования.

(Г) Формальные методы описания семантики обычно слишком сложны для рассмотрения пользователями языков.

(Д) Трансцендентальность семантики как онтологии субъекта.

4. Рассмотрим фрагмент программы, который предназначен для вставки элемента в связный список с возрастающими ключами:

```
...
type
  List = ^Node;
  Node = record
```

```
            Data : integer;
            Next : List
          end;
var
  p, q, Head : List;
  Key        : integer;
...
p := Head;
if p <> nil then
  while p^.Data < Key do
    p := p^.Next;
New (q);
q^.Data := Key;
q^.Next := p^.Next;
p^.Next := q;
...
```

Что произойдет, если фрагмент будет выполнен при условии, что значение переменной `Key` меньше значения любой из информационных частей списка?

(А) Новый узел будет вставлен в список случайным образом.

(Б) Новый узел будет вставлен после первого узла в списке.

(В) Новый узел будет вставлен перед последним узлом в списке.

(Г) Новый узел не появится в результирующем списке.

(Д) Структура списка будет разрушена.

5. Наиболее существенным вкладом объектно-ориентированного программирования является обеспечение эффективного способа:

(А) порождения эффективного кода;

(Б) повторного использования кода;

(В) автоматической генерации кода;

(Г) верификации порожденного кода;

(Д) загрузки программистов работой.

6. Укажите основные преимущества инкапсуляции:

1) более эффективное хранение данных;

2) более эффективное выполнение операторов;

3) более легкое чтение и понимание программ;

4) программы легче сопровождать.

(А) 1;

(Б) 2 и 3;

(В) 3;

(Г) 1 и 4;

(Д) 3 и 4.

7. Сопоставьте следующие термины объектно-ориентированного программирования с их наиболее подходящими определениями:

1) Класс	а) Совокупность данных и операций над ними
2) Сообщение	б) Описание набора однотипных объектов
3) Метод	в) Тело процедуры
4) Объект	г) Вызов процедуры

(А) 1 — а, 2 — б, 3 — в, 4 — г;

(Б) 1 — б, 2 — г, 3 — в, 4 — а;

(В) 1 — в, 2 — а, 3 — г, 4 — б;

(Г) 1 — в, 2 — б, 3 — г, 4 — а;

(Д) 1 — г, 2 — г, 3 — г, 4 — г.

8. Рассмотрим следующую грамматику в нотации BNF. Нетерминальные символы грамматики — '<s>', '<t>' и '<w>'. Терминальные символы — '@', '!', '#' и 'a' . Начальный символ — '<s>'.

```
<s> ::= <s> @ <t> | <t>
<t> ::= <w> ! <t> | <w>
<w> ::= # <w> | a
```

Грамматика определяет простой язык выражений с операторами `@`, `!` и `#` и операндом `a`.

Требуется привести все предложения языка (т. е. строки, состоящие только из терминальных символов), состоящие из не более трех символов. Какой из следующих результатов верен:

(А) a@a, a!a, ##a;

(Б) a, aa, aaa, a@a, a!a, #a, ##a;

(В) a, a@a, a!a, ##a;

(Г) a, a@a, a!a, #a, ##a;

(Д) a, @a, @@a, !a, !!a, #a, ##a.

9. Для грамматики из задания 8 определить, будут ли операторы @ и ! левоассоциативными (последовательность одинаковых операторов обрабатывается или группируется слева направо) или правоассоциативными?

(А) @ и ! — оба левоассоциативные;

(Б) @ и ! — оба правоассоциативные;

(В) @ левоассоциативный, а ! правоассоциативный;

(Г) @ правоассоциативный, а ! левоассоциативный;

(Д) оба оператора не ассоциативны, а асоциальны.

10. Для грамматики из задания 8 укажите приоритеты всех трех операторов в порядке их убывания:

(А) @ # !;

(Б) ! # @;

(В) @ ! #;

(Г) ! @ #;

(Д) # ! @.

О тестировании в Интернете

В ответах на технические вопросы кандидатам рекомендуется потренироваться на тестовых заданиях различных серверов в Интернете. Наиболее известным и популярным среди программистов является сервер компании BrainBench (**http://www.brainbench.com/**), тестирование на котором, к сожалению, теперь является платным.

Демонстрация технических разработок

Кандидат должен продемонстрировать несколько своих разработок и уметь ответить на любые технические вопросы по ним.

8.2.3.7. Проверка знания английского языка

Переводы как женщины: если верны, то некрасивы,
а если красивы, то неверны.
Моисей Сафир

Английский язык традиционно является языком международного общения программистов. На нем написано огромное количество документации, книг и статей, доступных в Интернете. В подавляющем большинстве случаев программистам приходится говорить, читать и писать на английском языке. Поэтому при собеседовании необходимо проверить способность кандидата вести диалог на английском языке, а также умение составить небольшой рассказ на заданную тему (техническую или бытовую). Кроме того, он должен перевести два небольших текста с русского языка на английский и наоборот, с английского на русский.

Пример технического текста для перевода

В качестве примера для перевода предложим текст, являющийся фрагментом стандарта языка C++ [ISO/IEC 14882:1998]:

"Ranking implicit conversion sequences.

Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion.

Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

- a conversion that is not a conversion of a pointer, or pointer to member, to bool is better than another conversion that is such a conversion;
- if class B is derived directly or indirectly from class A, conversion of B* to A* is better than conversion of B* to void*, and conversion of A* to void* is better than conversion of B* to void*;
- if class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B:
  - conversion of C* to B* is better than conversion of C* to A*;

```
[Example:
     struct A {};
     struct B : public A {};
     struct C : public B {};
     C *pc;
     int f(A *);
     int f(B *);
     int i = f(pc);       //  Calls  f(B *)
]
```

  - binding of an expression of type C to a reference of type B& is better than binding an expression of type C to a reference of type A&;
  - conversion of A::* to B::* is better than conversion of A::* to C::*;
  - conversion of C to B is better than conversion of C to A;
  - conversion of B* to A* is better than conversion of C* to A*;
  - binding of an expression of type B to a reference of type A& is better than binding an expression of type C to a reference of type A&;
  - conversion of B::* to C::* is better than conversion of A::* to C::*, and;
  - conversion of B to A is better than conversion of C to A.

Note: compared conversion sequences will have different source types only in the context of comparing the second standard conversion sequence of an initialisation by user-defined conversion; in all other contexts, the source types will be the same and the target types will be different".

Пример художественного текста для перевода

В качестве художественного текста предложим шуточный текст "Two Corporate Lessons".

- "Lesson Number One.
  - A crow was sitting on a tree, doing nothing all day. A small rabbit saw the crow, and asked him, "Can I also sit like you and do nothing all day long?"

- The crow answered: "Sure, why not?" So, the rabbit sat on the ground below the crow, and rested. All of a sudden a fox appeared, jumped on the rabbit and ate it.
- Moral of the story is: To be sitting and doing nothing, you must be sitting very, very high up.

❒ Lesson Number Two.

- A little bird was flying south for the winter. It was so cold; the bird froze and fell to the ground in a large field. While it was lying there, a cow came by and dropped some dung on it. As the frozen bird lay there in the pile of cow dung, it began to realize how warm it was.
- The dung was actually thawing him out! He lay there all warm and happy, and soon began to sing for joy. A passing cat heard the bird singing and came to investigate.
- Following the sound, the cat discovered the bird under the pile of cow dung, and promptly dug him out and ate him!
- The morals of this story are:

  1) Not everyone who drops shit on you is your enemy.

  2) Not everyone who gets you out of shit is your friend.

  3) And when you're in deep shit, keep your mouth shut".

8.2.3.8. Написание простой программы и ее анализ

Превосходно, Ватсон, Вы делаете успехи. Правда,
Вы упустили все существенные детали,
зато хорошо усвоили метод.

Артур Конан Дойль. "Приключения Шерлока Холмса"

Далее кандидату предлагается написать небольшую программу или фрагмент программы. Достаточно частым заданием является написание функции на языке C. Вот несколько простых заданий.

❒ Записать строку в обратном порядке.

❒ Переставить указатели в списке в обратном порядке.

❒ Выяснить, является ли список зацикливающимся, имея лишь два указателя на начало списка.

❒ Реализовать бинарный поиск.

❒ Найти максимальное по длине слово в строке.

❒ Проверить, является ли палиндромом строка, переданная параметром функции.

Предложим и более сложное задание, которое рекомендуется писать на командных языках утилит sed и awk. При этом информация по утилитам sed и awk должна быть доступна в виде `man` страниц или книги.

Имеется входной файл `tests_all` — список тестов с некоторыми параметрами запуска и входной файл со списком тестов в файле `tests_wrong`. Необходимо написать программу, которая будет формировать список строк в формате из `tests_all`, но только тех тестов, которые упомянуты в файле `tests_wrong`. Примеры файлов с исходными данными и результатом приведены в листинге 8.1.

Листинг 8.1. Пример файлов с данными к тестовому заданию

```
% cat tests_all
./tests/Exec4015320 Ver5.3 Intel
./tests/Exec4046953 Ver5.1 Intel
./tests/Exec4286851 Ver5.3 Intel Solaris
./tests/Exec4292089 Ver5.3 Intel
./tests/Exec4307633 Ver5.3 Intel Solaris
./tests/Exec4240772 Ver2.1 Intel
./tests/Exec4324693 Ver5.3 Intel
% cat tests_wrong
Exec4015320 has failed due to timeout
Exec4307633 bad, bad, very bad
Exec4324693 quite good, but lacks some info
% cat tests_result
./tests/Exec4015320 Ver5.3 Intel
./tests/Exec4307633 Ver5.3 Intel Solaris
./tests/Exec4324693 Ver5.3 Intel
```

8.2.3.9. Анализ чужого кода

В конце собеседования кандидату можно предложить выполнить анализ чужих программ. Например, после краткого просмотра программы в листинге 8.2 кандидат должен ответить на три вопроса:

1. Что должна делать программа "`prog.c`"?
2. Где ошибка в программе "`prog.c`"?
3. Можно ли заменить макросы "`GET_LONG`" и "`PLACE_LONG`" более быстрыми аналогами?

Листинг 8.2. Пример файлов к тестовому заданию по анализу чужого кода

```
% cat prog.h
#define PLACE_LONG(place, val)\
    do { ((char *)place)[0] = (char)(val);\
         ((char *)place)[1] = (char)((val) >> 8);\
         ((char *)place)[2] = (char)((val) >> 16);\
         ((char *)place)[3] = (char)((val) >> 24);\
         ((char *)place)[4] = (char)((val) >> 32);\
         ((char *)place)[5] = (char)((val) >> 40);\
         ((char *)place)[6] = (char)((val) >> 48);\
         ((char *)place)[7] = (char)((val) >> 56); } while (0)
#define GET_LONG(place)\
       (long long)(\
         ((unsigned char *)place)[0] |\
         (((unsigned char *)place)[1]<<8) |\
         (((unsigned char *)place)[2]<<16)|\
         (((unsigned char *)place)[3]<<24)|\
         (((long long)((unsigned char *)place)[4])<<32)|\
         (((long long)((unsigned char *)place)[5])<<40)|\
         (((long long)((unsigned char *)place)[6])<<48)|\
         (((long long)((unsigned char *)place)[7])<<56))
struct list_head {
       struct list_head *next, *prev;
};
% cat prog.c
#include <stdio.h>
#include "header.h"
int main() {
    struct list_head
    long long i = 1234567;
    char placeholder[8];
    PLACE_LONG(&placeholder, i);
    i = GET_LONG(&placeholder);
    printf("you've got: %ld\n", i);
    return 0;
}
```

8.2.3.10. Оценка результатов собеседования

Эрнст Резерфорд пользовался следующим критерием при выборе своих сотрудников.
Когда к нему приходили в первый раз, Резерфорд давал задание.
Если после этого новый сотрудник спрашивал,
что делать дальше, его увольняли.

Физический фольклор

Каждый из членов комиссии независимо оценивает профессиональные, личностные и интеллектуальные качества каждого кандидата (по некоторой шкале). На основе выставленных оценок вычисляется некий "условный балл". Можно просто суммировать все набранные кандидатом баллы, а можно ввести веса для критериев (возможно, специфичные для того проекта, для которого кандидат отбирается).

По окончании собеседования на основе оценок всех членов комиссии кандидату выставляется средняя оценка и выносится предварительное решение, которое, как правило, может быть одним из следующих:

- кандидат рекомендован для данного проекта;
- кандидат не рекомендован для данного проекта;
- кандидат может быть рекомендован с перечисленными замечаниями.

Можно проинтервьюировать несколько десятков кандидатов и выбрать одного, набравшего наибольшее количество баллов. Естественно, при этом список оцениваемых критериев для каждого кандидата должен быть одним и тем же. В дальнейшем эта оценка может быть полезна при отборе кандидатов в другие проекты.

О принятии окончательного решения о найме

Окончательное решение о найме кандидата должен принимать только менеджер проекта, потому что именно ему потом с ним работать. В том числе менеджер должен иметь право накладывать вето при необходимости. Нарушение этого правила ведет к найму по знакомству ненужных, слабых или неконтролируемых сотрудников.

8.2.3.11. О дополнительных домашних заданиях

Он был достаточно просвещен и мог задавать правильные вопросы.

Бхавад-Гита

В некоторых случаях для того, чтобы более точно оценить способности кандидата, который заинтересован в приеме на работу, ему предлагается небольшое домашнее задание на 2—3 дня. При необходимости, уточнение задачи проводится по электронной почте или телефону. Чем полнее вопросы и меньше звонков или писем, тем лучше. Существует два основных типа заданий для кандидатов.

- Разработка алгоритмов и программы на заданном языке программирования. Это задание предлагается кандидатам, рекомендуемым на должность инженеров, разработчиков программного обеспечения.

- Планирование и проектирование "в большом" некоторой серьезной задачи. Это задание для кандидатов, рекомендуемых в качестве руководителей программных проектов.

Пример домашнего задания

Дан файл с набором идентификаторов. Написать три программы (на языке C++):

- программу, которая печатает регулярное выражение, разбирающее все эти идентификаторы;
- программу, которая печатает наименьшее такое выражение;
- программу, которая генерирует анализатор таких регулярных выражений. Требования к сгенерированной программе:
  - ее код должен соответствовать стандарту ANSI C. Как следствие, он должен безошибочно компилироваться в исполняемый код;
  - полученная программа должна определять, подходит ли введенный идентификатор под полученное регулярное выражение.

О неточностях в постановке задачи

Обратим внимание, что в постановке данной задачи сознательно оставлены неточности и некорректности. Кандидат должен это отметить при работе над заданием.

8.2.4. Испытательный срок

Показывайте язык прежде, чем стисните зубы.

Станислав Ежи Лец

Испытание при приеме на работу имеет цель проверить соответствие работника поручаемой ему работе. В идеальном варианте, работник должен получить документ, включающий описание его функциональных обязанностей и прав. Если это не так, то этот документ должен быть составлен со слов нового руководителя. Такой документ особенно важен, если новая должность предполагает высокую степень ответственности и общение по служебной необходимости с широким кругом лиц. Еще одним полезным документом может стать схема внутренних и внешних контактов.

Будет полезным отправить начальнику по электронной почте составленное описание, схему и список задач на испытательный период. После внесения дополнений и изменений это описание становится отправной точкой.

В табл. 8.1 указаны процентные соотношения продолжительности испытательных сроков, которые обычно назначают работодатели (**http://www.osp.ru/cw/1998/47/73.htm**). В среднем около 10 % кандидатов не проходят испытательного срока и заменяются другими кандидатами.

Таблица 8.1. *Испытательные сроки, назначаемые работодателями*

Продолжительность испытательного срока	Количество назначений
1 месяц	48.3%
2 месяца	16.8%
3 месяца	31.7%
В зависимости от пришедшего кандидата	3.2%

Если Вы добрались до этого раздела (не в книге, а при реальном поиске работы), то нам остается только поздравить Вас и пожелать дальнейших успехов. Вы прошли достаточно серьезный путь и теперь должны наилучшим образом проявить себя в новой должности.

Если бы при приеме на работу к водителям относились так же, как к программистам

В завершении разд. 8.2 предложим шуточный текст. Если бы при приеме на работу к водителям относились так же, как к программистам, то требования могли бы быть выражены в следующем перечислении. От кандидата требуются: профессиональные навыки управления легковыми и грузовыми автомобилями, троллейбусами, трамваями, поездами метрополитена и фуникулера, экскаваторами и бульдозерами, спецмашинами на гусеничном ходу, боевыми машинами пехоты и современными танками, находящимися на вооружении стран СНГ и НАТО. Навыки экстремального вождения обязательны, опыт управления болидами "Формулы 1" приветствуется. Знания и опыт ремонта поршневых и роторных двигателей, автоматических и ручных трансмиссий, систем зажигания, бортовых компьютеров, антиблокировочных систем, навигационных систем (GPS) и автомобильных аудиосистем ведущих производителей обязательны. Опыт проведения кузовных и окрасочных работ приветствуется. Претенденты должны иметь сертификаты Mercedes, BMW, General Motors, а также справки об участии в крупных международных ралли не более чем двухлетней давности. Зарплата 1500—2500 рублей, определяется по результатам собеседования.

8.3. Карьера в области информационных систем

8.3.1. Карьера и ее планирование

Карьера — чудесная вещь, но она никого
не может согреть в холодную ночь.

Мэрилин Монро

Карьера — это стремление к созданию собственной значительности, продвижение в служебной или научной деятельности, достижение известности,

славы. Слово "карьера" происходит от французского слова "carriere", которое в свою очередь имеет предшественника в итальянском языке: "carriera" — "бег".

Более узко карьеру можно определить как продвижение человека по служебной лестнице и достижение им профессионального успеха. Карьера является элементом внутрифирменного управления и предполагает планомерные действия со стороны руководства и сотрудника, основанные на согласовании их интересов.

Напомним старинную поговорку: никакой ветер не будет попутным, если не знаешь куда плыть. Это справедливо и по отношению к выбору карьеры. Планирование карьеры в самом общем случае включает в себя следующие этапы.

- Определение цели.
- Определение круга обязанностей.
- Определение особенностей работы.
- Определение особенностей организации.
- Определение географических факторов.
- Определение перспектив работы.
- Оценку реальной возможности получить желаемую работу.

Не следует забывать и о том, что есть люди, которыми движут не карьерные соображения, в какой бы то ни было форме, а заинтересованность в результатах собственного труда и в самом процессе творчества. Творческая сторона профессиональной деятельности обычно и ведет к успеху, в том числе и к карьере.

Рассмотрим основные уровни (стадии) карьеры в области информационных систем (рис. 8.2). Карьера может развиваться как по профессиональной линии (технический специалист в программировании), так и по управляющей (менеджер программных проектов). Технические специалисты высокого уровня ценятся не менее, чем средний менеджмент, получая сравнимые зарплату и уважение.

Каждый уровень карьеры определяют следующие характеристики.

- Как много "надзора", руководства должно быть обеспечено представителю данного уровня со стороны представителей высшего уровня.
- Каким большим количеством информации и опытом обладает представитель данного уровня.

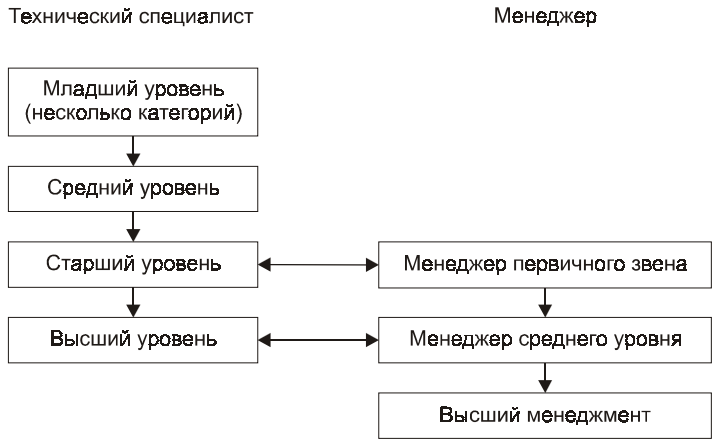


Рис. 8.2. Уровни карьеры

С точки зрения этих характеристик четыре технических уровня согласно [Conger 1994] выглядят так.

- *Младший* уровень. Над специалистом, относящимся к этому уровню, осуществляется постоянное руководство. Специалист обладает лишь основными навыками, и большая часть его рабочего времени уходит на обучение. Этот уровень проходится примерно за два года.
- *Средний* уровень. Большую часть времени специалист данного уровня работает самостоятельно, хотя с постоянным контролем выполняемой работы. Он обычно имеет достаточно хороший уровень знаний и опыт. Среднее время нахождения на этом уровне — от трех до пяти лет.
- *Старший* уровень. Специалист обладает широким диапазоном знаний и богатым опытом, является экспертом в теории и практике разработки программного обеспечения. Практически не требует руководства и руководит работой специалистов младших уровней. Как минимум три года требуется специалисту, чтобы пройти этот уровень. Однако для многих он становится завершающим в их карьере.
- *Высший* уровень. Это уровень технического лидера проекта.

К сожалению, любой уровень может стать завершающим в творческом развитии.

Существует три источника карьеры [Дубейковская 2000].

- *Потребность в стабильности.* Построение карьеры имеет явный парадокс — человеку приходится постоянно что-то менять, куда-то двигаться, но, тем не менее, в основе этого лежит ожидание стабильности и уверенности в будущем. Карьера представляет собой инвестирование личной энергии, денег и времени в будущую стабильность.

- *Потребность в признании.* Эквивалентами признания могут быть деньги, известность, ощущение всемогущества и даже страх подчиненных.
- *Решение "проблемы Чебурашки".* С точки зрения психологии, Чебурашка [Успенский 1977] — существо с проблемами идентификации, его переживания связаны с отсутствием внутреннего образа себя самого, непониманием того, как он выглядит не только в глазах других, но и в своих собственных. С подобными переживаниями сталкивается каждый человек. Карьера решает эту проблему таким образом, что человек идет по жизни, меняя вывески: "программист", "менеджер проекта", "президент компании" и т. п.

Три составные части карьеры могут быть охарактеризованы следующим образом [Дубейковская 2000].

- *Цель или направление движения.* Залог успешной карьеры — видение конечного результата и определение русла, по которому будет направлена личная энергия.
- *Профессиональный рост плюс личностное развитие.* Карьерист неизбежно столкнется с морально-этической стороной взаимоотношений с окружающими и его успех во многом зависит от того, насколько гармонично он сможет решить эту проблему.
- *Личный маркетинг.* Карьера объединяет в одном человеке личность, продающую товар, и собственно сам товар. Без соблюдения основных законов маркетинга в карьере практически нельзя обойтись.

Существуют различные точки зрения на верность профессии. В Японии, например, профессия обычно одна на всю жизнь, а в Америке менять профессии — дело совершенно естественное. Восток ценит верность прошлому, верность коллективу. Верность призванию на Востоке — верность рабочего места. Америка ценит освобождение из объятий всего, что не есть личность. Верность призванию означает смену рабочего места, чтобы опытным путем определить свою истинную цену, свое "я". Россия с Западной Европой находятся посередине в отношении к профессиональному выбору. Здесь ценится и верность, и подвижность, говорится и о призвании, и о выборе.

Существуют некоторые специфические особенности, влияющие на карьеру, присущие русскому народу. Во многом они обусловлены исторической спецификой [Психология 2000].

- При общем трудолюбии в труде наблюдается определенная *импульсивность*. Она выражается в смене приливов трудолюбия и азартной работы с периодами покоя и беззаботности. Этой черте трудно вписаться в размеренный четкий ритм графика разработки программного продукта. А черта эта исторически связана с особенностью сельского труда в России, предопределенного природно-климатическими явлениями, — коротким летом и длинной суровой зимой.

- *Небрежность в работе* исторически связана с условиями хозяйственной деятельности в русской общине. Если время на летние работы ограничено, то нерационально расходовать его, например, на постройку добротного моста через ручей.

Завершая этот раздел, напомним народную мудрость: первую половину жизни мы работаем на имя, потом оно работает на нас.

8.3.2. Совершенствование мастерства

Каждый человек, которого я встречаю, меня в чем-то превосходит,
и в этом смысле я могу у него поучиться.
Ральф Эмерсон

Для программистов, желающих совершенствовать свое мастерство, рекомендуются следующие методы:

- анализ конкретных практических примеров. Следует анализировать хорошие чужие программы, чтобы понять, как пишут другие программисты;
- изучение новых проблем, новых языков, нового инструментария и новых технологий. Это помогает программисту стать лучше и найти новые связи между тем, что ему уже известно;
- создание инструментальных программ, предпочтительно хороших, которыми пользуются другие.

Для студентов-программистов мы можем порекомендовать еще один метод — участие в олимпиадном (соревновательном) программировании, основные принципы которого мы рассмотрим в *разд. 8.3.3*.

Поддержка профессионального статуса и продолжающееся обучение — это необходимое требование для карьерного роста в области информационных систем. Знания должны уточняться, обновляться, развиваться, проходя спиралевидный путь, периодически возвращаясь к уже известным положениям, но уже на новом уровне.

Основными направлениями поддержки профессионального статуса являются:

- получение дополнительного образования. При получении дополнительного образования следует обратить внимание на те области программирования, которые были наиболее высоко оценены профессионалами *(см. разд. 1.3)*;
- чтение лекций в области специализации для совершенствования и систематизации знаний;
- участие в общинных разработках на основе открытых исходных текстов;
- работа в профессиональных организациях (например, IEEE);
- работа в пользовательских организациях (например, Java Users Group);

- чтение технической литературы. Насколько это важно — показывает результат опроса менеджеров в немецких фирмах (табл. 8.2). Так распределились ответы на вопрос: "Как вам удается следить за развитием в Вашей профессиональной области?";
- контакты с высшими учебными заведениями;
- посещение выставок;
- знакомство с отраслевыми обзорами.

Таблица 8.2. *Источники информации о развитии в профессиональной области*

Источники информации о развитии в профессиональной области	**Количество поддержавших**
Благодаря журналам и другой литературе по специальности	90 %
Благодаря специальным семинарам и заседаниям	58 %
Благодаря тематическим совещаниям	48 %

Мейлир Пэйдж-Джонс (Meilir Page-Jones) утверждает, что в разработке программного обеспечения существует 7 ступеней мастерства [Page-Jones 1990]. Эти ступени могут быть отнесены к методологии или технологии, а также языку или системе программирования. Поэтому то, к чему прилагаются эти ступени, обозначим "Ч". Перечислим эти ступени.

- *Наивный новичок*. Никогда не слышал о "Ч".
- *Осведомленный разработчик*. Прочел статью о "Ч", разобрался в общих чертах и может высказать свое мнение о достоинствах и недостатках "Ч".
- *Начинающий разработчик*. Посетил семинар, посвященный "Ч", успел пробежаться по небольшим примерам, иллюстрирующим возможности средства.
- *Практикующий разработчик*. Готов использовать "Ч" в реальном проекте.
- *Квалифицированный разработчик*. Постоянно использует "Ч" в своей работе. Очень недоволен, если лишается этой возможности.
- *Мастер*. Усвоил все детали "Ч" и знает, как обходить правила.
- *Эксперт*. Пропагандирует "Ч", пишет книги, выступает с докладами на конференциях.

Попробуйте оценить, на какой из ступеней по отношению к определенным знаниям находитесь вы. И сделайте определенные выводы самостоятельно.

8.3.3. Олимпиадное (соревновательное) программирование

8.3.3.1. История и принципы олимпиадного программирования

Человек вырастает по мере того, как растут его цели.
Ф. Шиллер

Олимпиадное программирование — это разновидность интеллектуального соревновательного программирования, на котором участники должны продемонстрировать свои способности в искусстве программирования в его классическом понимании. Участие в таких мероприятиях зачастую приносит много пользы не только начинающим программистам, но и имеющим многолетний опыт. Задачи, предлагаемые участникам таких соревнований, довольно часто берутся из реальных проектов и представляют собой как раз самую интересную и нетривиальную часть решения проблемы. И наоборот, алгоритмы, используемые при решении олимпиадных задач, могут быть применены в дальнейшем при разработке реальных приложений. Также полезно изучение стратегий поиска решений на основе командного подхода или обобщение методов тестирования и отладки, используемых в соревновательном программировании.

Начало крупным соревнованиям по программированию положил первый командный чемпионат мира среди сборных команд высших учебных заведений, проведенный под эгидой ACM (**http://www.acm.org**) в 1977 году. Этот чемпионат ежегодно проводится и по сегодняшний день (**http://icpc.baylor.edu/icpc/**). В первые годы в этих соревнованиях участвовали в основном команды североамериканских учебных заведений. Начиная с 1990 года, в связи с широким распространением персональных компьютеров, международный интерес к таким соревнованиям резко вырос. Европейская команда впервые поднялась на верхнюю ступеньку пьедестала почета в 1995 году. Чемпионом мира стала команда университета Альберта Людвига из немецкого города Фрайбург. В 2000 и 2001 годах чемпионат дважды выигрывала российская команда Санкт-Петербургского государственного университета. Способность страны собрать и воспитать команду (или даже несколько команд), готовую бороться за победу в финале чемпионата, свидетельствует об общем уровне компьютерных технологий в стране и ее возможном будущем потенциале в этом виде деятельности.

Достаточно популярно и другое международное соревнование — международная олимпиада по информатике для школьников, проводимая с 1989 года и имеющая личный (индивидуальный) характер. Для школьников также существует полезный информационный ресурс в Интернете: **http://neerc.ifmo.ru/school/**.

Обратим внимание на то, что фирма IBM ежегодно тратит десятки миллионов долларов на организацию олимпиад. Это позволяет ей привлекать к себе лучших студентов в области программирования и информационных технологий со всего мира.

Олимпиадное программирование имеет свои принципы, применяемые как для командных, так и для личных олимпиад.

- *Принцип простоты* заключается в том, что программа должна быть максимально простой. Чем проще программа, тем больше шансов, что она будет работать, да и отладить ее гораздо легче. Этот принцип помогает и при "реальном" программировании при разработке практически любого программного продукта. Например, он стоит на вооружении технологического подхода экстремального программирования и звучит в двух лозунгах так: "ищите самое простое решение, которое может работать" и "это вам не понадобится".
- *Принцип достаточности* утверждает, что не следует решать более сложную задачу, чем это необходимо. Например, если есть простой алгоритм, работающий за $O(n^3)$ и укладывающийся в заданные ограничения, то нет необходимости тратить дополнительное время на разработку сложного алгоритма за $O(n^2)$. Аналог этого принципа в "реальном" программировании заключается в том, что не следует тратить на проект больше ресурсов, чем это необходимо. С другой стороны, выбор неоптимального алгоритма может не устроить жюри или заказчика. Поэтому очень важно заранее точно понять, что именно требуется реализовать, а также оценить максимальное время работы программы.
- *Принцип свободного выбора* говорит о том, что любой из принципов может быть нарушен, если ситуация того требует. Этот принцип позволяет преодолевать неожиданные и нестандартные ситуации. Однако и проблем он вызывает больше всего, поскольку применять этот принцип на практике (да еще и так, чтобы от него не стало хуже) довольно непросто.

8.3.3.2. Специфика и стратегии командных соревнований

Командные соревнования предполагают свою специфику. Здесь на первый план выходят не только индивидуальные способности и таланты самих участников, но и их умение работать в коллективе. Если члены команды не умеют согласовывать свои действия, это может привести к результату даже худшему, чем если бы они разделились и выступали отдельно. Одна из технических проблем связана с тем, что на всю команду выделяется только одна ЭВМ.

В результате удача команды складывается из двух основных составляющих. Первая — это индивидуальное мастерство каждого ее члена. Список необходи-

мых навыков включает элементы математической культуры, знание основных структур данных, умение объяснить свои действия и алгоритмы коллегам. Не последнюю роль играют навыки быстрой отладки и приличная скорость набора текста программ. Вторая составляющая — умение оказывать друг другу помощь.

Существуют несколько стратегий олимпиадного программирования в команде, обычно состоящей из 3-х человек. Стратегии характеризуются тем, как основные операции по решению задач делятся между членами команды. Для примера рассмотрим четыре основные стратегии поиска решений на основе командного подхода.

- *Стратегия "с четким распределением ролей".* Во главе команды находится диктатор (капитан команды), являющийся генератором идей. Кроме него команда включает главного программиста и разработчика тестов, причем эти две роли могут меняться. Диктатор излагает решения задач, которые поступают на доработку к одному из участников. Главный программист занимается воплощением изложенных и доведенных до более или менее детального уровня идей в программу. К моменту окончания его работы над очередной задачей разработчик тестов должен уже иметь готовый набор тестов, позволяющий быстро проводить отладку и тестирование программы. После этого программа посылается жюри, а команда переходит к следующей задаче, идея которой уже проработана диктатором. Основное достоинство стратегии в том, что команда будет разумным образом управляться, поскольку есть капитан, полностью несущий ответственность за это. Однако этот чрезвычайно большой груз ответственности может стать и недостатком, т. к. каждая ошибка капитана может стать сокрушительным ударом по команде. На нем стыд и слава за все принятые решения, как по решению задач, так и по распределению работ. Такая стратегия является крайне упрощенной моделью команды, работающей над небольшим проектом и возглавляемой одним человеком. Также заметим, что эта стратегия очень близка к распределению ролей в бригаде главного программиста *(см. разд. 3.6.2.1)* с учетом удаления лишних для олимпиады ролей.
- *Стратегия "трое крутых".* Сразу после старта каждый участник выбирает задачу, с которой он в состоянии справиться, и начинает решать ее на отдельно взятом листочке. Член команды, выбравший самую простую задачу, сразу решает ее на ЭВМ. Для своей задачи каждый сам придумывает тесты, осуществляет отладку и сдает решение жюри. Эта стратегия может быть удобна для команд, состоящих из очень сильных участников. Недостатки стратегии заключаются в отсутствии общения между участниками во время игры. Например, команда может оказаться в цейтноте из-за неверного распределения работ, а любые попытки взаимопомощи приведут к существенным накладным расходам.

- *Стратегия "единое целое".* При таком подходе все члены команды решают все задачи вместе, начиная с простых, и постепенно переходят к сложным. Недостаток стратегии очевиден: отсутствие распараллеливания, уменьшающее общее число решенных задач. Эту стратегию обычно применяют команды, которые не борются за первое место, а просто хотят выступить достаточно хорошо, оказавшись в верхней части таблицы результатов.
- *"Гибкая" стратегия.* Эту стратегию применяла команда СПбГУ, дважды становившаяся чемпионами мира. Как ни парадоксально, стратегия команды заключалась в ее отсутствии. Точнее, стратегия каждый раз менялась и подстраивалась под состояние и настроение членов команды. Члены команды выполняли в различных ситуациях практически все описанные роли (диктатора, главного программиста, разработчика тестов). В зависимости от ситуации могло применяться парное программирование. В начале соревнования могла использоваться стратегия "трое крутых". Основная идея здесь заключалась в гибкости поведения. Именно гибкость позволила команде спокойно проходить через ситуации, которые вывели бы из строя любую другую команду.

Команда не сможет показать успешный результат, если она регулярно не тренируется. Тренеры команд очень много сил затрачивают на подготовку и организацию тренировок, которые надо проводить как минимум один раз в неделю. В результате тренировок вырабатываются многие навыки, которые полезны как в олимпиадном, так и в профессиональном программировании. Это чувство простоты реализации, умение видеть и чувствовать ошибки и потенциальные места их возникновения, умение строить контрпримеры к придуманному решению. И, конечно, умение планировать и работать в коллективе.

8.3.3.3. Примеры олимпиадных задач

Предлагаемые участникам олимпиад по программированию задачи можно условно разделить на две основные группы. Это задачи, к решению которых следует приступать, только глубоко продумав идею этого решения, и задачи, являющиеся не слишком сложными для человека, достаточно хорошо знающего языки программирования. В обязанность составителей входит и подготовка безнадежных задач, элементарное решение которых составителям неизвестно. Такие задачи очень часто представляют собой замаскированный условиями частный случай специальных задач, исследование которых требует определенных теоретических знаний. Они включаются, с одной стороны, для того, чтобы научить участников оценивать сложность задач и не решать их в первую очередь, а с другой стороны — позволить участникам проявить свой талант и реализовать неожиданные гениальные идеи.

В качестве примеров приведем три задачи, которые можно отнести к задачам первой группы, обладающих своими "изюминками". Вторая и третья задачи придуманы в центре подготовки одаренных программистов при СПбГУ, их автор — Николай Дуров.

Пример задачи с поиском экономии ресурсов

Дано $N \le 40$ монеток, каждая из них достоинством $M_i \le 200$ копеек. Требуется выделить из них две непересекающиеся подгруппы таким образом, чтобы суммы денег в первой и второй подгруппах были равны `M` и `M` было максимально возможным.

Формально эта задача описывается динамической системой `(k, A, B, R)`, где `A` — количество монеток в первой группе, `B` — количество монеток во второй группе, `R` — остаток, а `k` — количество уже рассмотренных (поделенных) монеток. Вспомним сначала классическую задачу про две кучки монеток, когда нужно разделить все монеты наиболее справедливо, таким образом, чтобы `|A - B|` → `min`. Функция, которую требуется пересчитывать, — `f(k, A, B, R)` → `boolean` представляет собой достижимость состояния `(k, A, B, R)`.

В предложенной задаче, как и в классической, допускается удаление одного из параметров. Удалим `R`, т. к. $A + B + R = S_1 + \ldots + S_k$ и, следовательно, `R` однозначно выражается через входные данные и все остальные параметры. Но на этом упрощения еще не закончились. Первый красивый прием состоит в удалении еще одного из параметров с помощью изменения функции. Поскольку из состояния `(A, B, k)`, очевидно, достижимо `(A, B, k+1)` (при `k < N`), то можно записать `g(A, B) = min k:f(A, B, k)`. В результате уже потребуется 200×200 памяти и $200 \times 200 \times 40$ времени.

Но это еще не предел. Подчеркнем, что нас интересует максимум значения `M` (т. е. `A = B = (A + B)/2` при равных группах), а количество `k` нас совершенно не интересует, только хотелось бы, чтобы `A – B = 0`. С помощью линейной замены параметров можно построить функцию `h(A – B) = max(A + B)`, которая при данном значении `A – B` и `h(0)` будет давать нам ответ на задачу, умноженный на 2. Это требует порядка 200×40 памяти и $200 \times 40 \times 40$ времени. Если бы ограничение (равное в нашем случае 200) было больше, это был бы существенный выигрыш.

Пример задачи с поиском простоты реализации

Дана "+/– машина", которая умеет выполнять следующие операции: добавить целое число, не превосходящее 10^6, в память, вывести *i*-е по значению число в памяти и удалить его, а также вывести всю память в порядке возрастания и очистить. Всего операций не более 500 000. Необходимо промоделировать работу этой машины.

Очевидно, что "наивное" решение этой задачи будет работать очень долго, но в арсенале хорошего программиста есть средство, которое позволяет такого

рода задачи решать за приемлемое время. Этим средством является алгоритм на основе сбалансированных деревьев (рандомизированных декартовых деревьев). Однако существует способ решить эту задачу еще в три раза быстрее (по фактическому времени работы) за счет простоты реализации.

Искомый метод заключается в следующем. Рассмотрим функцию промежутка `f(I)`, отображаемую в количество чисел, со значением, принадлежащим `I`, в памяти машины. Построим дерево, в котором у каждого промежутка `I` будут два сына: левая половина и правая, а корнем будет промежуток от 1 до 2^{20}. Такое дерево очень просто хранить в массиве, поскольку достаточно лишь организовать его, как известную структуру данных "куча":

1 — $[1..2^{20}]$

2 — $[1..2^{19}]$

3 — $[2^{19}+1..2^{20}]$

4 — $[1..2^{18}]$

...

2^{20} — $[1..1]$

$2^{20}+1$ — $[2..2]$

$2^{20}+2$ — $[3..3]$

...

$2^{20}+2^{20}-1$ — $[2^{20}..2^{20}]$

В листинге 8.3 приведены операции добавления и удаления числа в структуру "куча", а также вывод всех чисел из памяти.

Листинг 8.3. Операции работы со структурой данных "куча"

```
const M = 1 shl 20 - 1;
{ операция добавления числа x }
k := M + x;
while k > 0 do begin
    inc(a[k]); k := k shr 1
end;
{ операция удаления z-го по значению числа }
k := 1;
while k <= M do begin
    dec(a[k]); k := k * 2;
    if z > a[k] then begin dec(z, a[k]); inc(k) end
end;
```

```
dec(a[k]);
writeln(k - M);
{ вывод всех чисел из памяти, реализуемый как рекурсивная процедура }
procedure rec(x: integer);
begin
    if a[x] > 0 then begin
        if x > M then
            writeln(x - M)
        else begin rec(2 * x); rec(2 * x + 1) end;
    a[x] := 0
    end;
end;
```

Мы получили простые операции, гораздо более удобные для процессора, чем организация сбалансированного дерева. Все вычисления выполняются за логарифмическое время.

Пример задачи с математическим уклоном

Существуют задачи с математическим уклоном, который может быть хитро замаскирован. Пусть задано преобразование `z → (a * z + b)/(a - b * z)` и некоторое начальное значение `z0`. Через какое количество шагов последовательного применения этого преобразования `z` станет отрицательным?

Прямое моделирование здесь не поможет, т. к. число шагов может быть очень большим и произойдет потеря точности. Подход, связанный с возведением матрицы преобразования в степень, приведет к решению задачи за логарифмическое время. Однако есть и гораздо более красивый прием, который дает решение за константу. А именно, преобразуем выражение, вынеся за скобки в числителе и знаменателе `a` и сократив эту переменную. Результатом станет `(z + b/a)/(1 - b * z/a)`. Теперь выполним замену `b/a = c`, получив в итоге `(z + c)/(1 - c * z)`. Математик сразу обратит внимание на то, что это похоже на формулу для суммы тангенсов:

```
tg(alpha + beta) = (tg(alpha) + tg(beta))/(1 - tg(alpha)*tg(beta)).
```

Следующим шагом делаем замену `z → arctg(z)`. Наше преобразование стало линейным, и мы уже легко найдем требуемое число шагов.

Сделаем некоторые выводы, касающиеся олимпиадного программирования. С одной стороны, в нем явно прослеживаются черты любых соревнований, в том числе спортивных олимпиад. Из множества участников, стартовавших в начале, первые места и все почести достаются немногим. Однако есть и существенное отличие от спортивных соревнований: программисты используют

приемы, а не штампы. С другой стороны, существуют и открытые публичные конкурсы и олимпиады, проводящиеся на сайтах Интернета. Кроме того, все желающие могут испытать свои силы, потренировавшись на задачах, выложенных на сайтах Интернета, например **http://neerc.ifmo.ru/** или **http://contest.ur.ru/**.

8.3.4. Философия и стратегия программистских коллективов и организаций

> Чтоб мудро жизнь прожить, знать надобно немало.
> Два важных правила запомни для начала:
> ты лучше голодай, чем что попало есть,
> и лучше будь один, чем вместе с кем попало.
>
> *Омар Хайям*

Корпоративная культура — это исторически сложившиеся, присущие данной организации, устойчивые базисные образцы решения проблем внешней адаптации и внутренней интеграции. Выделяют следующие компоненты корпоративной культуры.

- *Нормы* — соглашения, регулирующие поведение людей и подразделений.
- *Ритуалы* — формальные или неформальные действия, усиливающие ощущение принадлежности к одной "семье". Существуют несколько видов ритуалов:
  - ритуалы *перехода* облегчают привыкание работника к новой социальной роли. Примером является ритуал представления нового сотрудника его будущим коллегам;
  - ритуалы *усиления* служат для повышения социального статуса работника. Примером может являться ежегодная церемония награждения сотрудников;
  - ритуалы *обновления* отражают деятельность в сфере обучения и развития. Примером может быть корпоративное соревнование сотрудников в профессиональной области;
  - ритуалы *интеграции* создают взаимную привязанность и добрые чувства сотрудников друг к другу. Простейшим примером будет корпоративная встреча Нового года.
- *Истории*, *легенды и мифы*, канонизирующие ранние достижения и мотивирующие сотрудников на новые свершения. Истории обычно посвящены героям и основателям компании и являются для остальных сотрудников образцом поведения. Во многих крупных программистских компаниях существуют истории о том, как их основатели в период становления компании кроме основной работы занимались погрузкой и разгрузкой компьютеров.

- *Физические символы*, направляющие внимание на некоторые специфические характеристики компании. В рабочих помещениях для команд, занимающихся проработкой идей будущих программных проектов, могут отсутствовать индивидуальные рабочие места. Такая особенность должна символизировать равенство и творческую атмосферу для инженеров.
- *Общий язык*, позволяющий членам команды взаимодействовать максимально эффективно.

Практически все крупные компании пропагандируют свою корпоративную культуру в книгах (например, "Билл Гейтс и сотворение Microsoft" [Ичбиа, Кнеппер 1997]) или, по крайней мере, на своих страницах в Интернете (например, информация компании Sun Microsystems, Inc. (**http://www.sun.com/aboutsun/coinfo/history.html**)). Рекламой своей корпоративной культуры компания может привлечь большое количество квалифицированных новых сотрудников.

Корпоративный дух — это состояние духовного единства и чувство взаимной поддержки между членами фирмы в достижении целей. Корпоративный дух — это тот идеал, к которому каждая компания должна стремиться.

8.3.5. Формы власти и стили руководства

Как хорошо, что дырочку для клизмы имеют
все живые организмы!
Николай Заболоцкий

Власть представляет собой реальную способность субъекта действовать или возможность влиять на ситуацию. Она отличается от *полномочий*, которые представляют собой право использовать ресурсы и сотрудников компании. Этот раздел дополняет информацию о процессе управления, который уже рассматривался в *разд. 3.2.2*.

Вертикальная координация (централизованное управление) — это соблюдение (или стремление к соблюдению) всех формальных правил бюрократической организации. Макс Вебер (Max Weber) сформулировал следующие основные признаки бюрократической организации [Цветаев 1999].

- Эффективность, достигаемая за счет строгого разделения обязанностей между членами организации.
- Строгая иерархия власти, имеющая форму пирамиды.
- Система правил и инструкций, обеспечивающих единообразие управленческой деятельности.
- Безличность административной деятельности и эмоциональная нейтральность отношений.
- Заинтересованность членов организации подчиняться и выполнять правила, мотивированные денежным вознаграждением и продвижением по службе.

О негативных последствиях бюрократизации

Один из наиболее известных недостатков бюрократической организации известен как закон Паркинсона: "Всякая административная единица склонна жить для самой себя и увеличивать как объем своих задач, так и число работников, необходимых для их выполнения".

Большинство крупных компаний по разработке программного обеспечения имеют вертикальную координацию власти.

Дискреционная власть — это власть исполнителя поступать по своему усмотрению. Она самостоятельно берется нижними подразделениями фирмы или работниками. Примеры такой власти можно наблюдать довольно часто.

- Власть секретаря руководителя.
- Авторитет опытного программиста.
- Возможность системного администратора повлиять на приобретение определенной аппаратуры.

Горизонтальная координация власти порождается децентрализацией, которая может быть вызвана рядом факторов (например, появлением филиалов, в том числе иностранных). Одна из наиболее характерных форм горизонтальных структур власти — временные программистские коллективы. Их руководители получают широкие полномочия, чтобы сосредоточиться на реализации некоторого важного проекта. При наличии в компании нескольких таких групп, структура их координации называется *матричной*.

8.3.6. Договорная основа трудовых отношений

Соотношение между временем, в течение которого работник работает, и рабочим временем есть величина постоянная и составляющая 0,6.

Дэниэл Макивор и Ослин Белл

Трудовой договор (контракт) — это двухстороннее соглашение между работником и работодателем (фирмой) об условиях обмена трудового вклада работника на его вознаграждение. Трудовой договор заключается в письменном виде и предполагает подчинение работника внутреннему распорядку фирмы, установленному работодателем. Единого универсального образца трудового договора (контракта) не существует. Однако имеется определенный перечень пунктов, которые обычно присутствуют в любых типах контрактов [Цветаев 1999].

- Имя нанимателя.
- Имя сотрудника.
- Название должности.
- Срок действия трудового контракта.
- Продолжительность испытательного срока (в случае необходимости).
- Размер заработной платы.

- Порядок и размер оплаты сверхурочных работ.
- Продолжительность рабочей недели.
- Часы работы.
- Время и продолжительность обеденного перерыва.
- Выходные и праздничные дни.
- Продолжительность ежегодного оплачиваемого отпуска.
- Государственное социальное страхование.
- Пенсионное обеспечение.
- Дисциплина труда.
- Порядок разрешения споров и разногласий.
- Порядок расторжения трудового контракта.

Мы рекомендуем работникам внимательно прочитать трудовой договор перед подписанием. В некоторых случаях оказывается полезным проконсультироваться с профессиональным юристом.

8.3.7. Заработная плата

Нельзя иметь все сразу, поэтому начни с малого — с денег.
Януш Васильковский

Одной из основных характеристик работы является заработная плата. На величину заработной платы в области информационных систем влияют многие факторы.

- Тип работы и уровень карьеры. Рассмотрим в качестве примера (табл. 8.3) среднюю заработную плату середины 90-х годов XX века нескольких специалистов разных уровней для штата Калифорния — самого высокооплачиваемого штата США в обсуждаемой нами области [Salary 1996].

Таблица 8.3. *Зависимость годовой зарплаты от уровня карьеры*

Специалист	Уровень карьеры	Величина зарплаты
Специалист по инженерии программирования	Младший уровень	41 200 $
	Старший уровень	59 500 $
Программист	Младший уровень	36 000 $
	Старший уровень	52 900 $
Менеджер проекта	Младший уровень	78 300 $
	Средний уровень	90 540 $

- Степень профессионализма в рамках одного уровня карьеры. В табл. 8.4 приведены диапазоны зарплаты на конец 90-х годов XX века для специалистов по инженерии программирования одной из процветающих компьютерных компаний штата Калифорния.

Таблица 8.4. *Диапазоны годовой зарплаты для уровней карьеры*

Уровень карьеры	Диапазон зарплаты
Младший уровень (1-я категория)	37 000—45 000 $
Младший уровень (2-я категория)	45 000—60 000 $
Младший уровень (3-я категория)	60 000—75 000 $
Младший уровень (4-я категория)	75 000—90 000 $
Средний уровень	90 000—120 000 $
Старший уровень	120 000—160 000 $

Об изменении средней заработной платы в кризисное время

Парадоксальная особенность была замечена в результате наблюдения за изменением заработной платы в США на границе двух тысячелетий. Во время, благоприятное для развития области информационных технологий, заработная плата росла незначительно. Однако в разгар так называемого кризиса в начале нынешнего века средняя заработная плата программистов возросла существенно. Это объясняется тем, что сокращение числа программистов шло за счет увольнения наименее квалифицированных и, следовательно, наименее оплачиваемых программистов. Кстати, учитывая возросшую нагрузку на оставшихся программистов, компании-работодатели часто дополнительно увеличивали их заработную плату.

- Размер и статус организации-работодателя.
- Характер работы — постоянная или контрактная. Специалисты, заключившие временный контракт (т. е. не являющиеся постоянными сотрудниками), получают на 30—60 % выше, но не имеют практически никаких социальных гарантий в компании.
- Страна проживания. В табл. 8.5 приведены диапазоны зарплаты специалистов по инженерии программирования, работающих в крупных компаниях Санкт-Петербурга (середина 2001 года).
- Географическое положение компании в стране. В табл. 8.6 приведены данные для штата Техас для тех же типов работы и уровней, что и для штата Калифорния (см. табл. 8.3). Зарплата, в среднем, ниже на 10—20 %.

Таблица 8.5. *Диапазоны месячной зарплаты для Санкт-Петербурга*

Уровень карьеры	Диапазон зарплаты
Младший уровень	300—1100 $
Средний уровень	900—2100 $
Старший уровень	1900—3100 $

Таблица 8.6. *Зависимость годовой зарплаты от уровня карьеры (пример другого географического положения)*

Специалист	Уровень карьеры	Величина зарплаты
Специалист по инженерии программирования	Младший уровень	37 600 $
	Старший уровень	52 500 $
Программист	Младший уровень	33 900 $
	Старший уровень	48 200 $
Менеджер проекта	Младший уровень	72 100 $
	Средний уровень	81 600 $

О зависимости заработной платы от географического положения компании в России

Тенденция зависимости заработной платы от географического положения компании в России выражена гораздо ярче по сравнению с США: в Москве зарплата в 2—3 раза выше, чем в Санкт-Петербурге. В других регионах России зарплата может быть в 2—5 раз ниже, чем в Санкт-Петербурге.

Итак, заработная плата зависит от очень многих факторов, в том числе от типа работы и степени профессионализма нанимающегося, статуса организации и ее географического положения. Мы рекомендуем внимательно изучить и сравнить предложения различных работодателей перед подписанием контракта. Особенно если Вы недавно переехали в другой город или страну. Кроме того, если Вы очень заинтересовали компанию-работодателя, она может быть готова принять Вас на очень большую начальную сумму. Главное для кандидата — вовремя понять это, как правило, на собеседовании.

Следует подчеркнуть, что размер заработной платы во многих случаях не является самой желанной целью для программистов. Другими, не менее важными целями могут быть:

- стремление получить творческое удовлетворение от создания программного обеспечения;
- ощущение полезности своего труда;
- стабильность работы и уверенность в завтрашнем дне;
- предоставление социальных услуг;
- правильно организованные и комфортные условия для работы;
- хорошие отношения в коллективе.

8.4. Почему некоторые программисты на порядок полезнее других?

> Объясните мне, пожалуйста, зачем нужно искусственно фабриковать Спиноз, когда любая баба может его родить когда угодно. Ведь родила же в Холмогорах мадам Ломоносова этого своего знаменитого. Доктор, человечество само заботится об этом и в эволюционном порядке каждый год упорно, выделяя из массы всякой мрази, создает десятками выдающихся гениев, украшающих земной шар.
>
> *Михаил Булгаков. "Собачье сердце"*

Приближаясь к концу данной книги, мы обратимся еще раз к вопросу, уже поднимавшемуся в *разд. 1.6.* Почему одни программисты оказываются на порядок полезнее других? В 1975 году, после выхода своей знаменитой книги, Брукс заставил многих задуматься, вынеся на широкую публику результаты некоторых исследований производительности труда в группе опытных программистов. Исследования показали, что "соотношения между лучшими и худшими результатами составили примерно 10 : 1 по производительности труда и 5 : 1 по скорости работы программ и требуемой для них памяти" [Брукс 1999]. Примерно через четверть века, в конце 1990-х, Картер и Сангер провели исследование, в котором вопрос ставился даже более жестко: "почему в программной инженерии некоторые люди на порядок, а то и два, более полезны, чем остальные?" (**http://progstone.nm.ru/**). Насколько и кому это нужно знать? В первую и основную очередь тем, кто управляет программным проектом, т. е. менеджерам. Именно перед ними встают задачи приема на работу и увольнения сотрудников, а также распределения работ в проекте. Во вторую очередь, многим начинающим программистам, которые ставят себе задачу добиться успеха и стать настоящими профессионалами. Наконец, опытным программистам, которых что-либо может не устраивать (например, величина заработной платы или отношения с руководством или коллегами).

8.4.1. Насколько корректна постановка проблемы?

Умненький, благоразумненький Буратино, хотел бы ты,
чтобы у тебя денег стало в десять раз больше?

Алексей Толстой.
"Золотой ключик или приключения Буратино"

Ответ на вопрос о полезности поможет нам решить и два "классических" вопроса: "что делать (чтобы стать полезнее)?" и "кто виноват (что некоторый программист менее полезен, чем его коллега)?". Если к ответу на первый вопрос мы попытаемся приблизиться в данной работе, то ответ на второй вопрос гораздо "больнее" и затрагивает честь и авторитет многих классиков. Например, существует нестандартная точка зрения, что именно книга Брукса [Брукс 1999], в которой, по мнению, высказанному на одной из страничек Интернета (**http://progstone.nm.ru/**) "каждое высказывание взято, как кажется, из практики блюдущих ритуал ISO 9000 зомби", привела к застою и кризису в производстве программного обеспечения.

"Полезный программист" — это программист, наилучшим образом пригодный для того, чтобы программировать. Соответственно, чем более он пригоден, тем более и полезен. Понятия "полезный" и "хороший" достаточно близки, и мы не будем далее их различать. Подчеркнем, что полезность является относительной, а не абсолютной величиной. Она во многом определяется тем конкретным проектом, которым программист занимается в данный момент.

Для начала попробуем разобраться в том, насколько деятельность программиста отличается от других видов деятельности. Нам придется опять задуматься над тем, кто такие программисты и что такое программирование, но более образно, чем мы делали это в *разд. 1.2*.

Работа программиста во многом заключается в передаче мыслей, образов, идей от человека к компьютеру. Известен простой вариант классификации профессий, в основу которого положен критерий того, на что или на кого направлена активная деятельность. Выделяют пять основных объектов профессиональной деятельности: человек, природа, техника, знаковая система и художественный образ. На их основе формируют профессиональные сферы, например "человек-человек" — это профессии, связанные с обучением, воспитанием, руководством и контролем. Конечно, профессиональная деятельность программиста тоже имеет определенный тип: "человек-знаковая система". Успешный специалист в данной сфере должен хорошо знать знаки, символы, любые знаковые системы. С другой стороны, с результатами работы программиста имеют дело другие люди, поэтому есть смысл говорить и о том, что программист близок к области "человек-человек".

Если говорить о категориях работников физического или умственного труда, то программистов следует относить скорее к работникам умственного

труда. Хотя здесь все далеко не так однозначно, и некоторые авторы предлагают ввести дополнительные категории, к которым можно было бы отнести и программистов, например "новые служащие". С производительностью работников физического труда все достаточно просто. Фредерик Тейлор (Frederick Taylor) изучил физический труд и разрушил его романтику, заменив "тайны ремесла" последовательностью элементарных движений. Производительнее работников физического труда делает знакомство с оптимальными способами исполнения и организации элементарных движений. Для физического труда основным является вопрос: "как задание должно быть выполнено?". Производительность работников умственного труда изучена гораздо хуже. Автор одной из наиболее интересных работ в этой области Питер Друкер (Peter Drucker) отмечает, что производительность работников умственного труда определяется шестью особенностями [Друкер 2003].

- Производительность работника умственного труда требует четкого ответа на вопрос: "*в чем заключается производственное задание*?".
- Ответственность за производительность целиком возлагается на самого работника. Работники умственного труда должны сами собой управлять, им нужна независимость.
- Непрерывная инновационная деятельность должна стать неотъемлемой частью умственной работы и включаться в производственное задание работника умственного труда.
- Работнику умственного труда надо, с одной стороны, постоянно учиться, а с другой — постоянно учить.
- Главным показателем работы является не количество или объем работы, а ее качество.
- Надо, чтобы работники умственного труда *хотели* работать на данную организацию и предпочитали этот вариант всем прочим возможностям.

Творчество является чертой работников умственного труда, в процессе творчества осуществляется поиск решения. Ремесленничество склонно к рутине. Оно скорее является чертой работников физического труда и лежит в основе реализации решения. Программисты хорошо знакомы с *противоречием между творческим характером поиска решения и рутинностью его воплощения*. Известно, что многие талантливые программисты не смогли довести найденное им решение до воплощения.

Попытки численно выразить степень полезности потребуют введения метрик. Идеальной метрики для оценки полезности программиста не существует. Однако мы предложим три основных, на наш взгляд, наиболее подходящих.

- Первую назовем *прагматичной метрикой*. Основная деятельность коммерческой организации обычно направлена на коммерческую прибыль.

Уточним, что будем далее называть свойством (feature) программного продукта его некоторую функциональность, за разработку которой заказчик готов платить деньги. Полезность программиста может быть выражена функцией от следующих параметров:

- времени, затраченного на написание программного свойства;
- количества (объема) программного кода. Здесь следует обращать внимание не на количество строк, а на количество реализованных свойств программного продукта;
- качества полученного кода (времени работы, затрат памяти, ошибок, содержащихся в свойстве). Керниган предлагает также отталкиваться от конечного продукта и отмечает, что "хорошие программисты пишут хорошие программы; плохие их не пишут. Так что выяснить это можно, к примеру, прочитав их тексты".

- Вторая метрика является *метрикой отношения к проекту*. Для программиста очень важным является оказаться на своем месте в проекте. Например, в новаторском проекте потребуются качественные программисты-ремесленники, которые будут доделывать мудрый, но корявый код талантливых программистов-идеологов. Общий случай может быть изложен следующим образом. Возьмем множество типов возможных задач и каждую из них снабдим весом, т. е. долей, с которой данный тип задачи встречается в реальной жизни. Для конкретного программиста вычисляем полезность для каждого типа задач и, суммируя с весами, получаем среднюю полезность программиста. Эта метрика напоминает нам, что иногда полезность в большой степени является ориентированной на проект или задачу. Программист, умеющий писать драйверы внешних устройств для редкой операционной системы, может оказаться очень полезным для ряда проектов. Однако в большинстве проектов такое умение не потребуется.
- Третья метрика является *метрикой характеристик хорошего программиста* и представляет собой функцию от совокупности индивидуальных характеристик программиста, характеристик его как члена коллектива программистов, а также характеристик рабочей среды, в которую он помещен. Характеристики хорошего программиста подробно рассмотрены в *разд. 8.4.2*.

8.4.2. Характеристики хорошего программиста

Дороже всего люди расплачиваются за то,
что пренебрегают банальными истинами.

Фридрих Ницше

Предложим схему одного из возможных подходов к пониманию того, кто есть хороший программист и почему он может быть лучше других

программистов. Сгруппируем наши знания о программистах в три основные группы.

- Программист как индивидуальный работник.
- Программист во взаимодействии с группой программистов.
- Программист в рабочем пространстве (окружении).

В каждой из этих областей программист должен обладать наилучшими качествами и характеристиками, выделяющими его среди коллег.

8.4.2.1. Программист как индивидуальный работник: личность, интеллект, опыт

Под *личностью* обычно понимают конкретного человека, обладающего сознанием и самосознанием и индивидуального по каким-либо своим свойствам. Некоторые черты и особенности мышления программистов мы уже обсуждали в *разд. 1.1.1.2*. Логично предположить, что чем большее количество упомянутых черт присутствует у программиста, тем он полезнее. Конечно, хороший программист должен обладать и многими другими чертами, например честностью и исполнительностью, трудолюбием и ответственностью.

Программирование — это очень быстро меняющаяся профессиональная сфера. Как следствие, вытекает такая важная черта программиста, как любознательность. Программирование не относится к тем профессиям, в которых можно достигнуть многого, не пополняя существующий запас знаний и не интересуясь новыми достижениями. Обязательными чертами являются способность к быстрому восприятию и обработке больших объемов информации. В ряде случаев на создание демонстрационного прототипа должны уходить считанные часы.

Определенные особенности накладывает и специфика конкретных выполняемых задач. Такие черты, как аккуратность и усидчивость, обычно требуются от специалистов по тестированию, а грамотность и наличие литературных способностей особенно важны для технического писателя.

Убедиться в наличии у программистов определенных черт в большинстве случаев можно лишь после длительного общения с ним. Однако иногда это сделать достаточно легко с помощью так называемого метода провокаций, который может применяться психологом при проведении собеседования с кандидатом. Кандидату предлагается листок, на котором записаны сложные термины из области программирования, среди которых присутствуют выдуманные, и просят подчеркнуть известные. Если подчеркнутыми оказываются и выдуманные слова, испытуемый может говорить неправду и в других ситуациях.

Важную роль играет физическое здоровье программиста, которое определяет его работоспособность. Причем под работоспособностью не обязательно

понимается сидение за компьютером сутки напролет. Хорошая работоспособность позволяет прокручивать идею в голове даже в то время, когда программист не сидит непосредственно за компьютером.

Интеллект представляет собой индивидуальные особенности, относящиеся к познавательной сфере. Умственные способности, особенности мышления, способность к обучению, логическому мышлению — это все то, что мы обычно относим к интеллекту. На интеллект программиста обращают внимание в первую очередь его менеджеры, в то время как коллегам он обычно интересен как личность.

Согласно теории интеллектуального порога, для овладения любой деятельностью нужен необходимый и достаточный уровень интеллекта. Если уровень интеллекта ниже, то человек не может работать, а если выше, то у него не будет прироста продуктивности.

> ***О креативности и интеллекте***
>
> Под креативностью понимают способность к порождению оригинальных идей и использованию нестандартных способов интеллектуальной деятельности. Обратим внимание на парадоксальный факт: высококреативные личности хуже решают задачи на репродуктивное мышление, к которым относятся практически все тесты интеллекта.

Опыт — это совокупность знаний и практически усвоенных навыков, умений. Программист, работающий в определенной предметной области, выполняет некоторые действия не задумываясь, на автоматизме. Опыт включает профессиональные навыки, знания специфических технологий и т. п. Критерием опытности еще около 10 лет назад было знание и владение языками C и C++, а сейчас это языки платформ Java и .NET. Естественно, опыт может распространяться и на редкие задачи, а также узкие предметные области.

Опытом определяется способность выделить существенное, например наиболее важные запросы в потоке поступающих. Опыт позволяет также структурировать знания, которые в свою очередь помогут правильному усвоению практической деятельности.

Знания также бывают разные. Например, знание языка программирования должно дополняться знанием базиса предметной области (т. е. того, что надо программировать). Отсутствие одного из них обычно приводит к невозможности программирования.

8.4.2.2. Программист во взаимодействии с группой программистов

Современное программирование является коллективным, и полезность отдельного программиста тесно связана с его полезностью для всей команды. Отношения программиста с окружающим коллективом могут рассматри-

ваться с разных точек зрения. Для коллектива человек интересен, если с ним приятно находиться рядом и разговаривать. С коммуникабельным и общительным человеком легко обсуждать подходы, проблемы, планы на будущее. Человек необщительный, с которым неприятно вести разговор, начинает выпадать из коллектива. Во многих проектах основная передача знаний между программистами происходит во время совместных чаепитий.

Существуют качества, специфичные для конкретных ролей программиста в команде. Их мы уже рассматривали в *разд. 3.6.2*. Замечательно, если в команде программистов есть человек, являющийся душой коллектива. Он поддержит дух разработчиков, сможет посоветовать, выслушать, поддержать неформальную коммуникацию, пригласит выпить чашку чая. Известны случаи, когда после увольнения таких людей (менеджерам казалось, что их личная производительность достаточно низка) производительность работ в проекте резко падала.

Заметим, что есть программисты, которые не могут работать в команде вообще, а также те, которые не могут работать в конкретной данной команде. Если они очень полезны для компании, то личное общение с ними может быть заменено общением по электронной почте.

8.4.2.3. Программист в рабочем пространстве (окружении)

Важным для программиста является умение создать себе рабочее место, способствующее повышению производительности его труда. Чем меньше число отвлекающих факторов, тем выше его производительность. Программист тратит 15—20 минут на то, чтобы подготовить мозг для творческого процесса и только после этого готов к созиданию. Любое отвлечение "сводит на нет" это время.

Скорее всего, сырая комната в подвале, где душно, темно и шумно и где стоит рабочий стол программиста, не прибавит тому энтузиазма в работе. Но тут многое зависит не только от самого программиста, а от того, насколько хорошие условия труда подготовлены менеджментом компании. Повысить производительность работы может даже автомат с напитками, возле которого могли бы собираться программисты. Общаясь, программисты быстрее находят решения своих проблем или просто отдыхают.

Большую роль играет наличие или отсутствие окон в рабочем помещении. Желательно наличие хотя бы одного окна, поскольку размышлять над проблемой некомфортно, если взгляд упирается в стену или в потолок. В некоторых компаниях сотрудника перемещают в офис с окнами через несколько лет работы, вне зависимости от статуса и продвижения по карьерной лестнице.

Обратим внимание на то, что многие компании поощряют работу своих сотрудников на дому. Это экономит ресурсы компании на размещение сотрудников, с одной стороны, и дает возможность программисту работать в привычной и комфортной домашней обстановке. Как правило, не менее

одного раза в неделю программисты собираются на встречу для обсуждения рабочих вопросов.

К характеристикам, повышающим производительность труда, относится также высокий уровень социальных гарантий и уверенность в завтрашнем дне, высокий уровень морали в компании. Немаловажную роль играет надежная семья, обеспечивающая духовное спокойствие. Известно, что хорошо работает человек, которому утром хочется на работу, а вечером — домой.

8.4.3. Можно ли стать полезнее?

Ученые из Новосибирского университета выяснили,
что мыши размножаются гораздо быстрее,
если им не мешают ученые
из Новосибирского университета.

Биологический фольклор

Ответ на вопрос о том, может ли программист стать полезнее, мы попробуем получить из введенных метрик. Предложим программисту действовать следующим образом:

- понять, выбрал ли он наиболее подходящую для себя область специализации, и изменить ее, если это не так;
- понять, обладает ли он характеристиками хорошего программиста, и стремится к их непрерывному улучшению;
- понять, руководит ли проектом, в котором участвует программист, компетентный и квалифицированный менеджер, и отказаться от участия в проекте, если это не так.

Как мы уже отмечали в начале *разд. 8.4*, работа над повышением производительности программиста начинается с того, что программист отвечает на вопрос: "в чем заключается мое производственное задание?". Логически близкими вопросами будут следующие: "что будет помогать мне выполнять задание?" и "что будет мешать мне выполнять задание?". Может потребоваться немало сил и времени, чтобы программисты смогли делать то, что им действительно положено делать и, собственно, за что они получают деньги. Естественно, одним из первоочередных действий здесь будет исключение посторонних дел из основной рабочей деятельности программиста.

Хорошей иллюстрацией к тому, как подходят разные программисты к решению вопроса "в чем заключается мое производственное задание?", является так называемая "гипотеза паковки и картостроения". Картер и Сандер (**http://progstone.nm.ru/**) в работе "Программистский камень" считают, что программирование — это *картостроение* (процесс исследования возможностей системы, природы задачи и желаний, а затем перенос результатов исследования на язык программирования). Картостроение — это стратегия,

альтернативная *паковке* — социально обусловленному общепринятому мышлению. Решение проблемы картостроителем приводит к красивым, компактным, элегантным программам, в которых нет места для ошибок. Практически существуют два отдельных состояния ума вокруг нас.

Установить, насколько хорошо личность умеет мыслить, достаточно сложно. Это можно сделать лишь в неторопливых беседах или после длительного рассуждения. Более того, способность мыслить часто пресекается взрослыми в детях, когда они пытаются размышлять и задают вопросы в попытке что-либо понять. В результате в среде программистов можно выделить две различные группы. Картостроители в основном придерживаются стратегии познания, как распространения и интеграции мысленных карт, а затем получения решения на их основе. Они быстро находят методы достижения своих целей, сверяясь со своими картами. Паковщики, оставаясь в большинстве, встают на путь накопления пакетов знаний. Их единственная цель — выполнить "правильное" действие.

Паковщиков можно отличить от картостроителей и по некоторым внешним признакам. Паковщики демонстрируют правила ведения дискуссий. Картостроителям во время дискуссий разрешается подпрыгивать и много кричать. В то же время, картостроителям должна быть предоставлена возможность сконцентрироваться во время работы, тогда как паковщики видят обычно работу, как состоящую из механической последовательности действий. Мы видим, что склад ума картостроителей в совершенстве позволяет им выполнять многие функции профессиональных программистов, особенно применение системного анализа.

8.5. Этические ответственности

8.5.1. Этика в работе программиста

Берестяная пенка дров.
Камин. Голландская зола ...
Спасал его Бенкендорф,
а любимая не спасла.

А. Вознесенский

Напомним еще раз о том, что *этика* — это учение о морали, ее сущности, структуре, функциях, законах, ее историческом развитии и роли в общественной жизни. В упрощенном понимании этику можно представлять как совокупность принципов, отделяющих правильное поведение от неправильного. Мораль регулирует поведение и сознание человека. Принципы морали распространяются на всех людей, определяя культуру межчеловеческих взаимоотношений. Над вопросами морали, так или иначе, приходится

задумываться всем и каждому. Моральная ответственность программистов, прежде всего, связана со следующими их чертами:

- компетентностью, которая включает понимание программистом реального уровня своих знаний и способностей;
- грамотностью в использовании ресурсов, включающей использование своего рабочего времени и вычислительной техники в целях выполняемого проекта;
- честностью, связанной с вопросами приобретения и владения интеллектуальной собственностью.

Кроме этого, программист должен разбираться в конфиденциальности, предполагающей неразглашение секретной внутренней информации компании.

В философских работах Декарта выведены правила морали, одно из которых, на наш взгляд, особенно интересно. Оно состоит в том, что нужно "всегда стремиться побеждать скорее себя, чем судьбу, изменять свои желания, а не порядок мира и, вообще, привыкнуть к мысли, что в полной нашей власти находятся только наши мысли". Данное правило предлагает соотношение между волей и разумом. Может показаться, что Декарт признает бессилие разума перед волей (судьбой). Однако он поясняет: "Наша воля по самой природе вещей стремится только к тому, что наш разум представляет ей так или иначе возможным". В какой степени программист обладает свободой воли и свободой выбора?

Программисты постоянно сталкиваются с проблемой выбора в различных ее вариациях. Огромное количество проблем связано с несогласием программиста с профессиональными действиями менеджера. Типичным источником разногласия может быть оценка менеджером времени на выполнение проекта. Программист может быть не согласен с решением менеджера о передаче недостаточно оттестированного программного продукта заказчику. Заметим, что корень таких проблем в том, что программист ориентирован на задачу, а менеджер — на успех.

Во многих случаях правильное решение проблемы найти очень трудно и перед вовлеченным в ее решение инженером встает этическая дилемма. Любой из альтернативных вариантов решения может стать нежелательным, т. к. может быть воспринят, как неэтичный. Для того чтобы легче было принимать решения в условиях неопределенности, можно принять некоторые простые правила и следовать им. Правила фиксируют некоторые нормы, принятые в компании, и образуют фирменные стандарты, составляющие часть корпоративной культуры. С точки зрения Сергея Валерьевича Сычева (**http://www.triz-ri.ru/themes/method/creative/creative28.asp**), правила сводимы к четырем простым принципам:

- если Вы упрощаете работу коллеге — Вы правы;

- если при решении сложных вопросов Вы спокойно следуете процедурам и фирменным стандартам, а не эмоциям и обидам — Вы правы;
- если то, что Вы делаете, защищает компанию — Вы правы;
- если Вы делаете больше, чем от Вас ожидали — Вы правы.

Поясним, например, первый принцип. Он следует из общего утверждения: все хорошо, что ведет к упрощению. Естественно, упрощение должно идти без ущерба для качества. Можно просто "сохранить файл", а можно "сохранить файл так, чтобы его мог найти коллега".

Одно из возможных правил, приводимых Сычевым, выглядит так: "Когда я работаю в команде, то понимаю, что, например, 10 : 00 — это 10 : 00, а не 10 : 15 и не 11 : 00. Я понимаю, что общая договоренность организует не только мое время, но и время моих коллег, а нарушение мной договоренности дезорганизует не только меня, но и коллег, которые в этом не виноваты. В случае различных накладок я всегда звоню и предупреждаю коллег заранее. Я рассчитываю на то, что и мои коллеги поступают аналогичным образом". Понятно, что следование таким правилам сильно упрощает жизнь большинству сотрудников компании.

Многие компании создают собственный корпоративный кодекс этики. Такой кодекс может объяснять сотрудникам, какого поведения ожидает от них компания. Существование кодекса является важным инструментом для формирования этических норм в организации. Профессиональные общества и институты также разрабатывают и публикуют кодексы профессионального поведения, например ACM/IEEE Code of Ethics (**http://www.acm.org/serving/se/code.htm**).

Некомпетентность программиста приводит к ошибкам в программном обеспечении, которые могут вызвать очень серьезные, иногда ужасные последствия. Известно много печально знаменитых катастроф, вызванных ненадлежащим функционированием программного обеспечения. Вот лишь два примера (**http://www.osp.ru/os/1998/06/21.htm**).

- Катастрофа ракеты-носителя Ariane-5 в 1996 году связана с некорректным повторным использованием модуля, для которого отсутствовала точная спецификация.
- Облучение смертельной дозой радиации на медицинском ускорителе Therac-25 в 1985–1987 годах связано с целым рядом причин, включая и программные (например, ситуацию типа гонки) и дефекты в постановке задач.

К катастрофическим последствиям в этих историях привело пренебрежение хорошо структурированными и стандартизованными технологиями программирования, которые должны опираться на высококвалифицированных программистов.

8.5.2. "Феномен науки" В. Ф. Турчина

Жизнь без начала и конца. Нас всех подстерегает случай.
Александр Блок

В процессе работы программистам приходится решать частные этические проблемы. Но решение их во многом зависит от того, как они отвечают для себя на несколько основных "глобальных" философских вопросов, над которыми обычно задумывается каждый разумный человек. Вот возможный набор таких вопросов.

- Что есть добро и что есть зло?
- К чему должен стремиться человек и чему противиться?
- Каковы смысл и цель существования каждого человека и всего человечества?

Многие из таких вопросов изучались известными философами. Иммануил Кант (Immanuel Kant) ставил вопрос: "что такое человек?" и выделял 3 подвопроса, последовательно пытаясь приблизиться к ответу: "что я могу знать?", "что я должен делать?" и "на что я могу надеяться?"

В этом разделе мы очень кратко рассмотрим подход выдающегося ученого Валентина Федоровича Турчина к построению этических и нравственных принципов. Турчин отмечает [Турчин 2000], что ответы на "глобальные" вопросы хочется получить с точки зрения науки. Но наука не может ответить на них без внесения дополнительных факторов: идей добра, цели, долга и т. п. Задачей науки является построение моделей той действительности, которая есть на самом деле. Поэтому она отвечает на вопросы типа: "что есть на самом деле?". Турчин считает, что любые попытки построения нравственных принципов на научной основе приводят к вопросу: "какова Высшая Цель?" (который тождественен вопросу "каково Высшее Благо?"). Из ответа на этот общий вопрос научные принципы позволят получить ответы на частные вопросы.

Стоит ли вообще обсуждать проблему Высшей Цели и собственно смысла жизни? Природа не дает человеку установок, он сам их себе создает. Для человека нет ничего естественного, кроме бесконечного развития. Нынешние духовные ценности лишь вехи на пути человечества. Следовательно, каждому из нас следует думать о смысле жизни. Думая об этом, мы создаем себе высшие установки. Это наивысший вид творчества, доступный человеку. И этот вид творчества будет нужен всегда, поскольку высшие установки будут меняться в процессе развития.

Что может зависеть от Вас лично? Насколько значительным может быть личный волевой акт отдельного человека? Человечество представляет собой большую и сильно связанную систему. Поступки одних людей самым серьезным образом влияют на поступки других. Такие системы обладают свойством расхождения траекторий, т. е. небольшие вариации в начальном

состоянии системы становятся все больше и больше. Кибернетической моделью этого является триггерный механизм — механизм со спусковым крючком. Требуются незначительные усилия, чтобы нажать на спусковой крючок, а последствия этого действия могут быть огромны. Следовательно, очень важно установить общие принципы, руководствуясь которыми, вероятность желательных последствий, т. е. добра, должна увеличиться.

Грустная нота: человек единственный из живых существ понимает, что он смертен. Мысль о неизбежности смерти создает мучительную ситуацию для него. Поэтому протест против смерти является источником, из которого этические учения черпают волевую компоненту. Все традиционные религиозные учения исходят из безусловной веры в бессмертие души. Протест против смерти используется как сила, заставляющая человека принять такое учение. Но человек, воспитанный на идеях современной науки, не может поверить в бессмертие души в традиционной религиозной формулировке. Воля к бессмертию ведет в этом случае к одной цели: внести свой личный вклад в космическую эволюцию. В книге "Феномен науки" Турчин подводит нас к основному выводу: "Высшая Цель человека — конструктивный вклад в эволюцию вселенной!".

Об основном противоречии социальной интеграции

В своих работах Турчин затрагивает множество интереснейших проблем. Например, о выборе между необходимостью включить человека в систему (непрерывно консолидирующееся целое) и необходимостью сохранить его как свободную творческую личность. Индивидуализм (присущий капитализму) и тоталитаризм (присущий социализму) — две противоположности, связанные общей цепью. Попытка разрешения проблемы заключается в том, что следует ставить своей задачей сознательную социальную интеграцию при сохранении и развитии творческой свободы личности.

8.5.3. Отступление "о свободе и стоимости программного обеспечения"

> Для меня, истинно и пламенно любящего живопись, не может быть лучшего желания, как положить начало общественного, всем доступного хранилища изящных искусств, принесущего многим пользу, всем удовольствие.
>
> *П. М. Третьяков*

Еще одна этическая проблема связана с противоречием между научным и коммерческим подходом к программным продуктам. Научный (академический) подход заключается в том, что результаты, производимые учеными, свободны и бесплатны. Коммерческий подход накладывает ограничения на свободу, подразумевает оплату за произведенные результаты и ехидно утверждает, что "бесплатный сыр бывает только в мышеловке". В результате программисты стоят перед еще одной этической дилеммой: идти по пути научному или коммерческому. Стоит уточнить, что в этом разделе речь пойдет о двух параметрах, которые не следует путать. Первый — это свобода программного обеспечения, а второй — его стоимость.

Степень свободы определяет, что разрешено и что запрещено делать с программным продуктом. Как правило, эта степень излагается в особом документе, называемом лицензией. *Лицензия* представляет собой договор, определяющий условия, на которых владелец авторских прав на программный продукт определяет условия его использования. Договор считается заключенным, если пользователь инсталлирует, копирует или осуществляет какой-либо иной доступ к программному продукту. Современные инсталляторы при установке продукта на компьютер обычно предлагают пользователю прочитать текст лицензионного соглашения и согласиться с его условиями. В случае отказа инсталляция прекращается. Лицензия может накладывать ограничения как на бинарный код, так и на исходные тексты программ.

В движении в направлении роста свободы программного обеспечения мы видим три основных этапа.

- *Движение за свободное программное обеспечение* (free software foundation). Основателем движения был профессор Ричард Столлман (Richard Stallman). Начало этого движения связано со стартом проекта GNU (**http://www.gnu.org/**) в 1984 году.
- *Движение за открытые исходные тексты* (open source development). Пропагандистом данного движения, являющегося уточнением (или "изменением метки") предыдущего, с 1998 года является Эрик Раймонд (Eric Raymond).
- *Движение за открытую проектную документацию* (foundation of open project documentation). Об организации этого движения было объявлено в 2002 году на торжественном открытии полуфинальных соревнований командного чемпионата мира по программированию ACM профессорами СПбГИТМО В. Г. Парфеновым и А. А. Шалыто (**http://is.ifmo.ru/?i0=works&i1=open_doc**). Данное движение является развитием предыдущих, но упор делается не на документацию программ, а на документацию проектов их создания.

Крупнейшей площадкой, предоставляющей инфраструктуру для разработки свободного программного обеспечения, является **http://sourceforge.net**.

Многим программистам движение за свободное программное обеспечение известно по двум основным проектам: GNU и Linux. В основе проекта GNU (что является аббревиатурой для рекурсивного имени GNU's Not Unix) лежит идея создания программной системы, полностью совместимой с Unix. Ричард Столлман изложил основные концепции проекта в документе "Манифест GNU" (**http://www.gnu.org/**). Он четко указал, что "золотое правило требует: если мне нравится программа, то я должен поделиться ею с другими, кому она тоже нравится". Огромной популярностью среди программистов является семейство компиляторов GCC (GNU Compiler Collection), распространяемое в исходных текстах и ориентированное на генерацию кода для многих архитектур ЭВМ.

Эрик Раймонд является пропагандистом концепции открытых программных текстов. Основные взгляды и достижения этой концепции изложены им в

эмоциональной статье "Собор и Базар" (**http://www.osp.ru/os/1999/09-10/071.htm**), основные положения которой мы уже обсудили в *разд. 3.6.3*. С некоторыми критическими взглядами на статью Раймонда можно познакомиться в работе Николая Безрукова "Повторный взгляд на Собор и Базар" (**http://www.linux.org.ru/books/GNU/misc/second-look-at-CatB.html**).

Дискуссии о том, хорошо или плохо распространять программы в открытых исходных текстах, периодически разгораются с новой силой. Особенно после того, как в результате утечек информации хакерам становится доступен исходный текст какого-либо крупного программного продукта, например операционной системы Windows. Да, с одной стороны, открытость кода облегчает хакерам поиск брешей и недостатков в системе и последующую их эксплуатацию. С другой стороны, закон Линуса гласит: "при достаточном количестве глаз все ошибки лежат на поверхности". На наш взгляд, это существенный довод в пользу программного обеспечения, распространяемого в открытых исходных текстах.

Обратим внимание на то, что идея открытой проектной документации возникала неоднократно. Она звучит еще в концепции грамотного программирования, предложенной Дональдом Кнутом *(см. разд. 3.2.4.3)*. Давно возникло понимание того, что ценность документации (например, описания архитектуры и интерфейсов) может значительно превышать ценность самих исходных текстов. Удачные примеры открытой проектной документации можно найти на сайте СПбГИТМО (**http://is.ifmo.ru/**).

Напомним, что второй интересующий нас в этом разделе параметр — это стоимость. Эта величина, как обычно, измеряется в денежном исчислении. Программный продукт может распространяться бесплатно, а может стоить ту или иную сумму, относясь к одной из следующих основных категорий.

- *Коммерческие программные продукты* (commercial software) распространяются за плату. Оплата производится двумя основными способами:
  - в момент получения копии программы. Так обычно оплачиваются коробочные программные продукты. Их использование до оплаты обычно не допускается;
  - после пробного использования в течение определенного срока. Так обычно оплачиваются распространяемые через Интернет программные продукты. На этот определенный срок пользователю предоставляется временный регистрационный ключ, который после оплаты заменяется постоянным.
- *Условно-бесплатные программные продукты* (shareware) могут быть использованы бесплатно в рамках определенного срока. Если по истечении срока пользователь захочет продолжить использование программы, он должен либо заплатить разработчику, либо добровольно удалить программный продукт со своего компьютера. Заметим, что, в отличие от коммерческих программных продуктов, многие условно-бесплатные продукты не откажутся работать и после истечения определенного срока.

- *Добровольно оплачиваемые программные продукты* (donationware) распространяются свободно. Однако их разработчик указывает пользователю, что тот, если захочет, может выслать ему некоторое вознаграждение. Однако это не обязательно.
- *Косвенно оплачиваемые программные продукты* (adware) характеризуются тем, что разработчик получает деньги от рекламодателей и включает демонстрацию рекламы во время работы программы.
- *Бесплатные программные продукты* (freeware) не требуют оплаты за их использование. Заметим, что благодаря таким продуктам легко распространять новые идеи и стандарты.

В качестве иллюстрации рассмотрим несколько примеров комбинаций параметров свободы и стоимости.

- Свободный и бесплатный программный продукт. В самом простейшем случае разработчик отказывается от всех авторских прав и распространяет продукт вместе с исходными текстами бесплатно. Любой желающий может вносить в тексты свои изменения и распространять свои программные продукты на основе исходного. Примером такой комбинации может служить графическая оболочка X Windows для операционной системы Unix.
- Свободный и платный программный продукт. Достаточно свободными лицензионными ограничениями обладают коммерчески поддерживаемые диалекты операционной системы Linux. Фактически, с такими продуктами можно делать все, а плата берется за их предварительную подготовку.
- Несвободный и бесплатный программный продукт. Разработчик обладает всеми авторскими правами и накладывает ограничения на использование продукта, но распространяет его бесплатно. Примером такой комбинации может служить инструментарий JDK для платформы Java.
- Несвободный и платный программный продукт. Практически все коммерческие программные продукты распространяются с существенными лицензионными ограничениями. Примером такой комбинации является операционная система Microsoft Windows.

8.6. Заключение

8.6.1. Выводы и рекомендации

Кто прожил много лет, тот много узнал, и подчас противоречивого.

Талейран

Сложность управления программными проектами отражена во множестве мифов, один из которых мы сейчас приведем. Однажды программист, гуляя по побережью Финского залива, обнаружил древний кувшин, открыл его, и выпустил на волю джинна. Джинн предложил выполнить любое желание программиста, а тот попросил построить автомобильный мост над океаном

до Америки, поскольку летать самолетами боялся. Джинн решил, что задание программиста будет весьма сложно выполнить технически, учитывая огромное расстояние и глубину океана, и предложил выбрать любое другое желание. Немного подумав, программист попросил сделать его менеджером, который бы всегда успешно управлял программными проектами. В свою очередь джинн ненадолго задумался и потом спросил: "так ты хочешь 2-х или 4-х полосный мост до Америки?".

Кризис программирования (*см. разд. 1.6*) продолжается, а проектирование и разработка программных продуктов по-прежнему являются нетривиальной деятельностью. Индустрия привлекала в ряды программистов многих людей, обещая, что программировать — это легко. Когда у начинающих программистов возникали сомнения, им давали новый набор инструментов, затем новый язык программирования, методологию или технологию. Обычно менеджеры при этом уделяли мало внимания центральному и ключевому фактору — фигуре самого программиста. В этой главе мы напомнили очевидную мысль о том, что успех любого проекта определяется людьми (программистами), в нем участвующими. А воздействие на людей в организации оказывают два основных фактора:

- иерархическая структура организации с основным средством воздействия — вертикальной иерархией власти;
- корпоративная культура с вырабатываемыми организацией совместными ценностями, социальными нормами, установками поведения, регламентирующими социальные нормы.

Говоря об отечественных программистах, отметим, что существует два их основных поколения. Границу между ними провела так называемая "перестройка" в начале 1990-х годов. Каждое из поколений характеризуется своей спецификой мотивации.

- Основной мотивацией программиста до перестройки были статус и карьера. Самый продуктивный программистский возраст на тот момент был от 30 до 45 лет. У программистов были мощные теоретические знания и обладание двумя важными качествами: фантазией и способностью усваивать новый материал. Заметим, что первый доступ к компьютеру тогда программисты получали примерно в 20-летнем возрасте.
- Основной мотивацией постперестроечного программиста стали заработная плата и, в меньшей степени, карьера. Снижение среднего возраста разработчиков — отличительная черта современного программирования. Достаточно часто студент учебного заведения может включиться в реальный производственный процесс.

Большая часть выводов и рекомендаций для менеджеров уже была сделана в тексте данной главы. Предложим еще несколько тезисных утверждений.

- Обращая внимание на значимость и роль отдельных программистов, одновременно подчеркнем важность независимости процесса ведения проекта от поведения отдельных его исполнителей.

- Возрастает роль независимых разработчиков-консультантов в области создания программного обеспечения. Это согласуется с одним из предсказаний Скотта Адамса (Scott Adams), автора серии сатирических комиксов, на тему некомпетентности менеджеров: "В будущем квалифицированные профессионалы станут сбегать из штата фирм и все чаще делаться своими собственными боссами. Они превратятся в предпринимателей, консультантов, вольных стрелков, проституток и карикатуристов" [Адамс 2002].
- Менеджеры должны осторожно относиться к энтузиастам, освоившим новый язык или систему программирования "за 21 день" или "за 9 $^1/_2$ недель". В большинстве случаев они еще не готовы к серьезной работе.
- Чтобы удерживать деятельность компании на высоком нравственном уровне, недостаточно разработать кодекс этики. Лидеры компании своим примером должны поддерживать этические ценности.
- Большая часть вины за возможные проблемы, возникающие при разработке программных продуктов, лежит исключительно на менеджерах проектов. Именно в их компетенции были такие задачи, как правильный подбор и подготовка исполнителей-программистов, продумывание их мотивации, планирование проекта и многое-многое другое.

Если руководство компании уделит особое внимание подбору и назначению грамотных и квалифицированных менеджеров проектов, то могут быть решены многие кризисные проблемы. Компетентный менеджер создаст профессиональную команду и будет управлять ею, как дирижер оркестром. В силах менеджера навести порядок в работе программистов и организовать слаженную работу команды.

8.6.2. Рекомендации по литературе

Три основные настольные книги профессионального менеджера программных проектов, уже не раз упоминавшиеся в нашей работе, таковы:

- "Мифический человеко-месяц или как создаются программные системы" [Брукс 1999];
- "The Psychology of Computer Programming" [Weinberg 1998];
- "Peopleware" [DeMarco, Lister 1987].

Основные подходы к менеджменту также излагаются в следующих работах:

- "Основы менеджмента" [Овсянко 1999]. Книга дает общее представление о современном состоянии теории и практики менеджмента, в том числе кадрового;
- "Психология менеджмента" [Психология 2000]. Учебник прослеживает психологизацию менеджмента в процессе его эволюции. Рассматриваются психологические особенности найма сотрудников в организации;

- "Особенности национальной карьеры" [Ушаков 1999]. Книга написана живо и интересно, с привлечением опыта большого количества менеджеров по персоналу российских и иностранных компаний;
- "Время — деньги. Создание команды разработчиков программного обеспечения" [Салливан 2002]. В книге рассказывается о реальном опыте успешной разработки коммерческого программного обеспечения в начинающей компании и о том, как выбрать и сохранить нужных специалистов;
- "Как сделать карьеру с помощью Интернета" [Соломенчук 2002]. В книге рассматриваются аспекты использования Интернета при поиске работы. Приводятся наиболее важные статьи трудового законодательства.

"Рекомендации по преподаванию информатики в университетах (Computing Curricula 2001: Computer Science)" включают тему "Социальные и профессиональные вопросы" (Social and professional issues). Основные разделы этой темы:

- управление программными проектами;
- социальный контекст информатики;
- профессиональные и этические ответственности.

8.6.3. Вопросы и задания

Предложим небольшое количество вопросов и заданий, которые предназначены для проверки знаний в области менеджмента программных проектов.

1. Приведите примеры конкуренции между компьютерными компаниями на мировом рынке. Каковы их формы, используемые средства, прямые и косвенные последствия?
2. Проанализируйте систему измерения результатов в компьютерной компании, деятельность которой Вам знакома.
3. Приведите примеры реального внутреннего и внешнего вознаграждения, которое имело место в известных Вам компьютерных компаниях.
4. Основная общая цель организации, выражающая причину ее существования, обозначается как ее миссия. Коротко сформулируйте миссию нескольких известных Вам компьютерных компаний.
5. Приведите пример влияния плохих коммуникаций на результаты деятельности компьютерной компании.
6. В какие формальные и неформальные группы входите Вы, работая над проектом в компьютерной компании?
7. Проанализируйте систему планирования и контроля в компьютерной компании, деятельность которой Вам знакома. Как можно исправить отмеченные Вами недостатки?
8. В чем должны заключаться особенности мотивации студентов, работающих над программным проектом (параллельно с учебой)?

9. Составьте объявление о приглашении на работу в Вашу компанию инженера по тестированию.
10. Были ли в Вашей профессиональной деятельности ситуации, когда Вы стояли перед этической дилеммой? Решение дилеммы было принято на основе собственных или корпоративных этических стандартов?
11. Мы надеемся, что эти вопросы, как обычно, дадут Вам пищу для размышлений, а результаты размышлений в свою очередь породят очередные вопросы. "Снежный ком" знаний будет расти, но будет расти и та неизвестность, с которой он соприкасается своей поверхностью. А это означает, что в Ваших руках скоро окажется следующая книга!

ЛИТЕРАТУРА

> Лучшая книга по программированию для непрофессионала — "Приключения Алисы в стране чудес", поскольку это лучшая книга для непрофессионала в любой области.
>
> *Алан Дж. Перлис*

1. *Адамс 2002* — Адамс С. Будущее по Дилберту. — Минск: ООО "Попурри", 2002.
2. *Брукс 1999* — Брукс Ф. Мифический человеко-месяц или как создаются программные системы. — СПб.: Символ-Плюс, 1999.
3. *Викентьев 1998* — Викентьев И. Л. Приемы рекламы и public relations. — СПб.: Изд. дом "Бизнес-Пресса", 1998.
4. *Друкер 2003* — Друкер П. Задачи менеджмента в XXI веке. — М.: Издательский дом "Вильямс", 2003.
5. *Дубейковская 2000* — Дубейковская Я. С. Стоп. Кадры!: Управление персоналом для умных. — Екатеринбург: Изд-во Уральского университета, 2000.
6. *Ичбиа, Кнеппер 1997* — Ичбиа Д., Кнеппер С. Билл Гейтс и сотворение Microsoft. — Ростов-на-Дону: Изд-во "Феникс", 1997.
7. *Карнеги 1991* — Карнеги Д. Как завоевывать друзей и оказывать влияние на людей. — М.: Прогресс, 1991.
8. *Кэролл 1990* — Кэролл Льюис. Приключения Алисы в стране чудес. — М.: Наука, 1990.
9. *Носова, Козадеров 1998* — Носова И. А., Козадеров Н. П. Программное обеспечение: правовые проблемы, пути их решения. — М.: КомпьютерПресс, 1998.
10. *Овсянко 1999* — Овсянко Д. В. Основы менеджмента: Учебное пособие. — СПб.: Изд-во Санкт-Петербургского университета, 1999.
11. *Пиз 1992* — Пиз А. Язык жестов. — Воронеж: НПО "МОДЭК", 1992.
12. *Психология 2000* — Психология менеджмента: Учебник // Под ред. проф. Г. С. Никифорова. — СПб.: Изд-во Санкт-Петербургского университета, 2000.
13. *Салливан 2002* — Салливан Э. Время — деньги. Создание команды разработчиков программного обеспечения. — М.: Издательско-торговый дом "Русская Редакция", 2002.

14. *Соломенчук 2002* — Соломенчук В. Г. Как сделать карьеру с помощью Интернета. — СПб.: БХВ-Петербург, 2002.

15. *Турчин 2000* — Турчин В. Ф. Феномен науки: кибернетический подход к эволюции. — М.: ЭТС, 2000.

16. *Успенский 1977* — Успенский Э. Н. Крокодил Гена и другие сказки. — М.: Детская литература, 1977.

17. *Ушаков 1999* — Ушаков А. А. Особенности национальной карьеры. — М.: ЗАО "Бизнес-школа "Интел-Синтез", 1999.

18. *Ходяченко 1999* — Ходяченко В. Б. Маркетинг: Учебное пособие. — СПб.: Изд-во Санкт-Петербургского университета, 1999.

19. *Холопова, Лебедева 1995* — Холопова Т. И., Лебедева М. М. Протокол и этикет для деловых людей. — М: ИНФРА-М, 1995.

20. *Цветаев 1999* — Цветаев В. М. Кадровый менеджмент: Учебное пособие. — СПб.: Изд-во Санкт-Петербургского университета, 1999.

21. *Clifton 1995* — Mitchell H. Clifton. Self-Assessment Procedure XXIII: Programming Languages // Communication of the ACM. Vol. 38, No. 5. May 1995.

22. *Conger 1994* — Conger, Sue A. The New Software Engenering. Wadsworth Publishing Company, 1994.

23. *DeMarco, Lister 1987* — DeMarco Tom, Lister Tim. Peopleware. Dorset House Publishing, 1987.

24. *ISO/IEC 14882:1998* — ISO/IEC 14882: 1998 Programming language C++. International standard.

25. *Leveson, Turner 1993* — Nancy G. Leveson, Clark S. Turner. An Investigation of the Therac-25 Accidents // Computer, July, 1993, pp. 18—41.

26. *Page-Jones 1990* — Meilir Page-Jones. The Seven Stages in Software Engineering // American Programmer, July-August 1990.

27. *Salary 1996* — Salary Survey. How much is your job worth? // Future Outlook, 1996.

28. *Weinberg 1998* — Gerald Weinberg. The Psychology of Computer Programming. Dorset House Publishing, 1988.

Все кончено, — слабым голосом сказал кот и томно раскинулся в кровавой луже, — отойдите от меня на секунду, дайте мне попрощаться с землей.

М. Булгаков. "Мастер и Маргарита"

Предметный указатель

А

Абстрагирование 252
Автоматизированные обучающие системы 384
Адресация 493
Алгоритм 58
✧ анализ 59
✧ построение 59
✧ теория 59
Анализ требований 136
Архитектура:
✧ CISC 409
✧ EPIC 411
✧ RISC 410
✧ SPARC 419
✧ VLIW 411
✧ аккумулятор 407
✧ набора команд 397
✧ параллельная 434
✧ регистр-регистр 407
✧ сети ЭВМ 439
✧ стековая 408
✧ фон Неймановская 406
✧ ЭВМ 397
Атомарные транзакции 508

Б

База данных 353

В

Версия:
✧ альфа 180
✧ бета 180
✧ первой поставки пользователю 180
✧ разработчика 180
Виртуальный офис проекта 323
Вычислительная система:
✧ организация 397
✧ функциональное описание 418

Г

Грамматика 292

Д

Данные 253
Декомпилятор 333
Диаграмма:
✧ вариантов использования 147
✧ деятельности 148
✧ классов 146
✧ компонентов 144
✧ кооперации 144
✧ объектов 147
✧ последовательности 148
✧ потоков данных 145
✧ развертывания 145
✧ состояний 147
✧ сущность-связь 143
✧ функционального моделирования 145
Диалоговые системы 382
Динамические системы поддержки принятия решений 324
Драйвер 526

Ж

Жизненный цикл:
✧ программного обеспечения 61
✧ программы 111

З

Зависимость:
✧ по данным 264
✧ по управлению 264
Знак:
✧ денотат 238
✧ значение 238
✧ форма 238
Знание 279
✧ декларативное 280
✧ процедурное 280

И

Инженерия программирования 33
Инкапсуляция 84
Инструментальные системы 369
Инструментальная среда программирования 369
Интегрированная среда 371
Интеллектуальные вопросно-ответные системы 383
Интервальная арифметика 413
Интернет 445
Интернет-технологии 445
Интерпретатор 333
Интерфейс:
✧ графический пользовательский 320
✧ командной строки 320
Информатика 31
Информационные системы руководителя 324
Искусственный интеллект 354

К

Карты класс-ответственность-кооперация 146
Карьера 538, 565
Качество 218
✧ достаточно хорошее программное обеспечение 223
✧ модель:
  • CMM 221
  • SPICE 222
Класс 147
Классификация 56
Команда:
✧ версии 66
✧ локализации 66
✧ разработчиков компонента 65
✧ разработчиков продукта 65
Коммутаторы 435
Компетентность 26
Контрольные точки 180
Корпоративная культура 578
Кризис программирования 53

М

Маршрутизация 496
Менеджмент 123
✧ история и эволюция 127
Метод:
✧ абстрактных типов данных 83
✧ алгоритмической декомпозиции сверху вниз 98
✧ аппликативности 88
✧ восходящего проектирования 142
✧ единообразия 92
✧ изменения состояний 80
✧ модульной организации частей программы 99
✧ настраиваемости 88
✧ нисходящего проектирования 142
✧ объектно-ориентированной декомпозиции 83
✧ описательной модели вычислений 95
✧ пересылки сообщений 83
✧ проектирования предметных областей 142
✧ расширения ядра 142
✧ рекурсивного поведения 88
✧ синхронизации исполняемого кода 100
✧ структурного кодирования 99
✧ унификации 92
✧ управления потоком исполнения 80
Методология программирования 32
✧ автоматного 106
✧ аспектно-ориентированного 107
✧ в ограничениях 95
✧ доступ-ориентированного 107
✧ императивного 79
✧ логического 91
✧ нейросетевого 107
✧ объектно-ориентированного 83
✧ параллельного 100
✧ структурного 98
✧ управляемого потоками данных 107
✧ управляемого событиями 107
✧ функционального 87

Моделирование 60
Модель 60
✧ данных 277
✧ реляционная 278
Модуль 265
✧ рутинность 269
✧ связность 267
✧ сцепление 268
Модульное программирование 264
Монитор 481
Мотивация 126

Н

Наследование 84

О

Обозреватели 349
Образованность 26
Объект 147
Оператор:
✧ безусловного перехода 99
✧ ветвления 260
✧ исключения 262
✧ цикла 261
Операционная система 455
✧ распределенная 461
✧ реального времени 463
✧ сетевая 461
✧ централизованная 460
✧ ядро 456
Определение 56
Основы программирования 32
Открытые системы 28
Отладчик 341
Оффшорное программирование 214

П

Память 416, 514
✧ виртуальная 517
✧ распределенная общая 519
Платформа 67
✧ аппаратная 67
✧ инструментальная 68
✧ операционная 67
✧ сетевая 67
✧ языковая 68
Поисковые системы 364
Полиморфизм 255
✧ включения 84
Пользователь 64
Понятие 56
Поток управления 467
Прагматика 239
Предметная область 142
Программа 57, 244
✧ жизненный цикл 111
Программирование 31
✧ исследовательское 203
✧ модульное 264
✧ олимпиадное 571
✧ оффшорное 214
✧ структурное 257
✧ теоретическое 31
Программное обеспечение 57
✧ жизненный цикл 61
Программное средство 57
Программный инструмент 316
Программный комплекс 57
Программный продукт 57
✧ архитектура 138
✧ интернационализация 302
✧ лицензия 597
✧ локализация 303
Проект 62
✧ одностраничное описание 119
Проектирование 136
✧ архитектуры 136
✧ детальное 136
✧ паттерн 192
Протоколы 491
✧ стек TCP/IP 492
Профессионализм 25
Профиль 224
Процесс 111, 465
✧ анализ требований и проектирование 136
✧ ввод программы в действие 166
✧ возникновение и исследование идеи 117
✧ завершение эксплуатации 170
✧ отладка 162
✧ сопровождение 167
✧ стандартный:
  • поставка 171
  • приобретение 171
  • разработка 172
  • сопровождение 172
  • эксплуатация 172

✧ тестирование 161
✧ управление 122
Процессор 416
Псевдокод 144
Психология:
✧ программирования 539
✧ управления 539

P

Разработка программ:
✧ авторская 205
✧ бригада главного программиста 208
✧ иерархическая модель 207
✧ коллективная 206
✧ матричная модель 207
✧ общинная 213
✧ типы совместной деятельности 213
✧ ядерная модель 209, 210
Редактор связей 515
Резюме 542
Рекомендации по преподаванию информатики 35
Репозиторий 372
Рефакторинг 335

C

Семантика 239
Семафор 478
Семиотика 239
Сигналы 471
Синтаксис 239
Системы:
✧ автоматизированные обучающие 384
✧ анализа корректности программного кода 332
✧ верификации программ 368
✧ генерации трансляторов 329
✧ деинсталляции 345
✧ диалоговые 382
✧ динамические поддержки принятия решения 324
✧ документирования 351
✧ инсталляции 343
✧ инструментальные 369
✧ искусственного интеллекта 354
✧ информационные руководителя 324
✧ открытые 28
✧ отслеживания проблем 342
✧ поддержки работы виртуальных групп 380
✧ поисковые 364
✧ построения срезов 340
✧ программирования 315
✧ разработки интерфейсов 353
✧ синхронизации удаленных пространств 380
✧ управления:
  • базами данных 353
  • версиями файлов 377
  • компиляцией и построением программ 336
  • рабочими пространствами 378
✧ формального преобразования программ 368
Собеседование 547
Сообщение 148
Спецификация 137
✧ средства 137
✧ язык 137
Средства:
✧ автоматизации разработки программ 370
✧ динамического построения профиля 339
✧ исследования и просмотра текста с гипертекстовыми возможностями 351
✧ обнаружения плагиата 349
✧ отслеживания тестового покрытия 339
✧ поиска в тексте на основе шаблонов 347
✧ поиска различий в тексте 347
✧ работы с текстом на основе регулярных выражений 345
✧ реинжиниринга 344
✧ сборочного программирования 368
Стадия 112
Стандарт 224
Стиль программирования 153
✧ грамотного 160
✧ защитного 159
✧ правило стандартизации 153
Структурное программирование 257
✧ визуальное 258

T

Текстовый редактор 349
✧ синтаксически ориентированный 350
Теоретическое программирование 31

Тестирование:
✧ автоматическое 163
✧ детерминированное 163
✧ динамическое 163
✧ дымовое 163
✧ интеграционное 163
✧ модулей 163
✧ подсистем и систем 163
✧ процедур и функций 163
✧ ручное 163
✧ случайное 163
✧ статическое 163
✧ структурное 163
✧ функциональное 162
Тестовое покрытие 165
Тестовый монитор 338
Технологический подход 112
✧ адаптивная разработка 200
✧ итеративная разработка 197
✧ каскадно-возвратный 182
✧ каскадно-итерационный 182
✧ каскадный 181
  • с перекрывающимися процессами 183
  • с подпроцессами 184
✧ кодирование и исправление 181
✧ компьютерный дарвинизм 204
✧ конкретизирующее программирование 192
✧ модель процессов MSF 187
✧ постадийная разработка 197
✧ рациональный унифицированный процесс 185
✧ сборочное программирование 190
✧ семейство Crystal 200
✧ синтезирующее программирование 190
✧ спиральная модель 184
✧ технология стерильного цеха 192
✧ формальное конкретизирующее программирование 195
✧ формальное сборочное программирование 195
✧ формальное синтезирующее программирование 195
✧ эволюционное прототипирование 196
✧ экстремальное программирование 198
Технология программирования 32, 111
Тип данных 253
✧ абстрактный 254
✧ контроль 256
✧ полиморфный 255
Транслятор 326
Тупик 485

У

Удаленный вызов процедур 503
Усложнители декомпиляции 335
Устройства ввода-вывода 418
Утилита 316

Ф

Фаза 116
Файл 522
Формат чисел IEEE 754 412

Х

Хранилища данных 323
✧ витрины 323
✧ интранет 324
✧ корпоративные 323

Ш

Шина 417

Э

Электронная библиотека 362
Электронная почта 322
Электронный календарь 322
Этика 592

Я

Язык 240
✧ естественный 298
✧ моделирования 243
✧ моделирования данных 278
✧ неформализованный 243
✧ программирования 244
✧ формализованный 243
Языки программирования:
✧ в ограничениях 96
✧ динамически типизируемые 256
✧ императивного 81
✧ логического 94

- объектно-ориентированного 86
- параллельного 102
- предметная область:
  - агенты 290
  - конфигурирование 291
  - описание аппаратуры 291
  - промежуточные 289
  - разметка 288
  - создание виртуальной реальности 291
  - создание графики 290
  - сценарии 289
  - форматирование 288
- семейство:
  - Ada 285
  - C 284
  - Forth 287
  - Lisp 286
  - Pascal 285
  - Prolog 287
  - Simula 285
  - параллельных 284
  - универсальных 282
  - уникальных 283
- статически типизируемые 256
- структурного 99
- функционального 90
- характеристики 244