Александр Леоненков

Самоучитель UML 2

Санкт-Петербург

«БХВ-Петербург»

2007

УДК 681.3.06
ББК 32.973.26-018.1
Л47

Леоненков А. В.

Л47 Самоучитель UML 2. — СПб.: БХВ-Петербург, 2007. — 576 с.: ил.

ISBN 978-5-94157-878-8

Рассмотрена современная технология объектно-ориентированного анализа и проектирования программных систем и бизнес-процессов в контексте нотации унифицированного языка моделирования UML 2. Подробно изложены все понятия языка UML 2 в полном соответствии с оригинальной спецификацией последней версии этого языка. Приведены конкретные рекомендации по разработке канонических диаграмм языка и рассмотрены особенности разработки моделей с помощью CASE-средства Borland® Together® Designer. Описана нотация OCL — языка объектных ограничений, по которому практически отсутствует информация на русском.

Для системных и бизнес-аналитиков, архитекторов программ, руководителей проектов и информационных служб, корпоративных программистов и студентов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.12.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-878-8

© Леоненков А. В., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление

Предисловие 1

Структура книги 3
Рекомендации по изучению языка UML 4
Благодарности 5
Постскриптум 6

ЧАСТЬ I. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ 7

Глава 1. Базовые принципы и понятия технологии разработки объектно-ориентированных информационных систем 9

1.1. Основные понятия моделирования систем и программных приложений 10
1.2. Методология объектно-ориентированного анализа и проектирования 15
1.3. Концепция разработки архитектур, управляемых моделями 22
1.4. Основные этапы развития UML 2.0 25

Глава 2. Основные элементы нотации языка UML 2.0 34

2.1. Назначение языка UML 2.0 34
2.2. Общая структура языка UML 2.0 38
2.3. Пакеты в языке UML 2.0 42
2.4. Основные пакеты метамодели языка UML 2.0 44
2.4.1. Пакет *Абстракции* 45
2.4.2. Пакет *Основы* 45
2.4.3. Пакет *Конструкции* 46
2.4.4. Пакет *Простейшие Типы* 47
Boolean (Логический) 47
Integer (Целочисленный) 48
String (Строка) 48

UnlimitedNatural (Неограниченное натуральное число) 49
2.4.5. Пакет *Модели* 50
2.5. Особенности спецификации метамодели языка UML 2.0 51
2.6. Особенности изображения диаграмм в нотации UML 2.0 60
2.7. Механизмы расширения в языке UML 2.0 65
2.7.1. Стереотип 66
2.7.2. Ограничение 67
2.7.3. Помеченное значение 69

ЧАСТЬ II. ДИАГРАММЫ ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ ЯЗЫКА UML 2.0 71

Глава 3. Диаграмма вариантов использования (use case diagram) 73

3.1. Диаграмма вариантов использования — исходная концептуальная модель проектируемой системы 73
3.1.1. Назначение диаграммы вариантов использования 74
3.1.2. Субъект вариантов использования 75
3.2. Основные графические элементы диаграммы вариантов использования 76
3.2.1. Вариант использования 77
3.2.2. Актер 79
3.2.3. Комментарий 82
3.3. Отношения на диаграмме вариантов использования 83
3.3.1. Отношение ассоциации 84
3.3.2. Отношение включения 86
3.3.3. Отношение расширения 88
3.3.4. Отношение обобщения 92
3.3.5. Пример диаграммы вариантов использования для системы продажи товаров в интернет-магазине 94
3.4. Формализация функциональных требований к системе с помощью диаграммы вариантов использования 97
3.4.1. Классификация требований в модели FURPS+ 97
3.4.2. Спецификация функциональных требований с помощью текстовых сценариев 98
3.4.3. Пример сценария для системы продажи товаров в интернет-магазине 100

Глава 4. Диаграмма классов (class diagram) 104

4.1. Диаграмма классов — основная логическая модель проектируемой системы 104
4.2. Класс 107
4.2.1. Имя класса 110
4.2.2. Атрибуты класса 112
Вид видимости 114
Кратность 116
4.2.3. Операции класса 117
4.2.4. Параметр 120
4.3. Отношения между классами 123
4.3.1. Ассоциация 124
4.3.2. *N*-арная ассоциация 130
4.3.3. Ассоциация-класс 131
4.3.4. Квалификатор 133
4.3.5. Обобщение 134
4.3.6. Множество обобщения 137
4.3.7. Агрегация 140
4.3.8. Композиция 142
4.3.9. Зависимость 144
4.3.10. Реализация 145
4.4. Интерфейс 145
4.5. Шаблон 148
4.6. Диаграмма классов для системы продажи товаров в интернет-магазине 150

Глава 5. Диаграмма композитной структуры (composite structure diagram) 154

5.1. Композитная структура 154
5.2. Композитный класс 156
5.2.1. Часть 156
5.2.2. Соединитель 158
5.2.3. Роль в спецификации экземпляра класса 160
5.3. Порт класса 161
5.4. Кооперация 165
5.5. Применение кооперации 167
5.6. Шаблон кооперации 170

Глава 6. Дополнительные диаграммы структуры 175

6.1. Диаграмма пакетов 175
6.1.1. Пакет 176

6.1.2. Зависимость пакетов 178
6.1.3. Импорт пакета 179
6.1.4. Импорт элемента 180
6.1.5. Слияние пакетов 183
Общие правила слияния пакетов 187
Правила для пакетов 188
Правила для классов и типов данных 189
Правила для свойств 189
Правила для ассоциаций 190
Правила для операций 191
Правила для перечислений 191
Правила для ограничений 192
6.2. Диаграмма объектов 194
6.2.1. Объект 194
6.2.2. Спецификация экземпляра 195
6.2.3. Слот 197
6.2.4. Значение экземпляра 199

Глава 7. Диаграмма последовательности (sequence diagram) 201

7.1. Диаграмма последовательности — основная модель взаимодействия элементов проектируемой системы 202
7.2. Линия жизни 205
7.3. Сообщения и сигналы 207
7.3.1. Сообщение 208
7.3.2. Сигнал 211
7.4. Комбинированный фрагмент 213
7.4.1. Альтернативы (*alt*) 216
7.4.2. Утверждение (*assert*) 217
7.4.3. Завершение (*break*) 218
7.4.4. Критический регион (*critical*) 219
7.4.5. Рассмотрение (*consider*) 220
7.4.6. Игнорирование (*ignore*) 220
7.4.7. Цикл (*loop*) 222
7.4.8. Отрицание (*neg*) 223
7.4.9. Необязательный (*opt*) 224
7.4.10. Параллельный (*par*) 225
7.4.11. Слабое следование (*seq*) 225
7.4.12. Строгое следование (*strict*) 226
7.5. Специальные фрагменты и элементы взаимодействия 226
7.5.1. Использование взаимодействия 226

7.5.2. Декомпозиция части 228
7.5.3. Инвариант состояния 230
7.5.4. Продолжение 232
7.5.5. Шлюз 234
7.6. Специальные ограничения на диаграммах последовательности 235
7.6.1. Временное выражение 236
7.6.2. Временное событие 236
7.6.3. Действие наблюдения времени 237
7.6.4. Интервал 238
7.6.5. Временное ограничение 239
7.6.6. Продолжительность 239
7.6.7. Действие наблюдения продолжительности 240
7.6.8. Ограничение на продолжительность 241

Глава 8. Диаграмма деятельности (activity diagram) 245

8.1. Концептуальные основы моделирования деятельности 245
8.1.1. Деятельность и действие 246
8.1.2. Узлы и дуги деятельности 247
8.1.3. Семантика деятельности 251
8.1.4. Семантика действия 252
8.2. Узлы управления 254
8.2.1. Начальный узел 254
8.2.2. Узел финала деятельности и потока 255
8.2.3. Узел решения 256
8.2.4. Узел слияния 258
8.2.5. Узел разделения 259
8.2.6. Узел соединения 260
8.3. Специальные действия 262
8.3.1. Действие передачи сигнала 262
8.3.2. Действие приема события 263
8.4. Узлы потока объектов 265
8.4.1. Узел объекта 265
8.4.2. Центральный буфер и хранилище данных 267
8.4.3. Входные и выходные контакты объектов 269
8.4.4. Узел параметра деятельности 271
8.4.5. Множество параметров 273
8.5. Специальные регионы 275
8.5.1. Разбиение деятельности 275
8.5.2. Регион прерываемой деятельности 278
8.5.3. Обработчик исключения 280

Глава 9. Вспомогательные диаграммы взаимодействия 285

9.1. Диаграмма коммуникации (communication diagram) 286
9.1.1. Линия жизни 288
9.1.2. Связь 289
9.1.3. Сообщение 290
9.1.4. Формат записи сообщений 292
9.1.5. Модель коммуникации 295
9.2. Диаграмма обзора взаимодействия (interaction overview diagram) 298
9.3. Временная диаграмма (timing diagram) 301
9.3.1. Основные элементы временной диаграммы 302
9.3.2. Первая форма временной диаграммы 304
9.3.3. Вторая форма временной диаграммы 305
9.3.4. Третья форма временной диаграммы 306

Глава 10. Диаграмма конечного автомата (state machine diagram) 308

10.1. Концептуальные основы моделирования конечных автоматов в языке UML 2.0 309
10.2. Простое состояние 314
10.2.1. Секция имени 315
10.2.2. Секция внутренней деятельности 315
10.2.3. Секция внутренних переходов 316
10.2.4. Отложенные события 317
10.3. Псевдосостояния 317
10.3.1. Начальное псевдосостояние 318
10.3.2. Узел завершения 319
10.3.3. Выбор 319
10.3.4. Соединение 321
10.3.5. Разделение 322
10.3.6. Слияние 323
10.3.7. Точка входа 324
10.3.8. Точка выхода 324
10.3.9. Неглубокая история 325
10.3.10. Глубокая история 327
10.3.11. Финальное состояние 329
10.4. Переход 329
10.4.1. Сторожевое условие 330
10.4.2. Переходы завершения и события завершения 331

10.4.3. Составные переходы 332
10.4.4. Передача сигнала 333
10.4.5. Прием сигнала 333
10.4.6. Действия на переходе 334
10.4.7. Правила разрешения и срабатывания переходов 335
10.4.8. Конфликтующие переходы 337
10.5. Композитные состояния и регионы 339
10.5.1. Основные определения 339
10.5.2. Вход и выход в простом композитном состоянии 341
10.5.3. Вход и выход в ортогональном композитном состоянии 344
10.5.4. Скрытая секция декомпозиции 345
10.6. Состояние подавтомата 347
10.7. Протокольный конечный автомат 350
10.7.1. Протокольное состояние 351
10.7.2. Протокольный переход 352

Глава 11. Диаграмма компонентов (component diagram) 356

11.1. Особенности физического моделирования в языке UML 2.0 356
11.2. Компонент 360
11.3. Интерфейс 364
11.4. Порт 366
11.5. Соединитель 368
11.5.1. Собирающий соединитель 368
11.5.2. Делегирующий соединитель 370
11.6. Зависимость 373
11.7. Реализация 375
11.8. Стереотипы компонентов 376

Глава 12. Диаграмма развертывания (deployment diagram) 382

12.1. Узел 384
12.1.1. Среда выполнения 385
12.1.2. Устройство 387
12.2. Артефакт 388
12.3. Спецификация развертывания 390
12.4. Отношения на диаграмме развертывания 392
12.4.1. Развертывание 393
12.4.2. Манифестация 395
12.4.3. Путь коммуникации 396
12.5. Стереотипы узлов 397

ЧАСТЬ III. АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ НОТАЦИИ UML 2.0 И CASE-СРЕДСТВА BORLAND® TOGETHER® DESIGNER 2005 401

Глава 13. Особенности реализации графической нотации языка UML 2.0 в среде Borland® Together® Designer 2005 403

13.1. Общая характеристика CASE-средства Borland® Together® Designer 2005 404
13.2. Особенности рабочего интерфейса Borland Together Designer 2005 406
13.2.1. Главное меню 407
13.2.2. Стандартная панель инструментов 408
13.2.3. Окно проекта 410
13.2.4. Окно навигатора модели 411
13.2.5. Окно навигатора диаграмм 412
13.2.6. Окно инспектора 413
13.2.7. Окно диаграммы 414
13.2.8. Стандартная панель инструментов окна диаграммы модели 415
13.2.9. Специальная панель инструментов диаграммы модели 417
13.2.10. Окно истории 418
13.3. Назначение операций главного меню 419
13.3.1. Пункт меню *File* (Файл) 419
13.3.2. Пункт меню *Edit* (Редактирование) 421
13.3.3. Пункт меню *Search* (Поиск) 422
13.3.4. Пункт меню *View* (Вид) 422
13.3.5. Пункт меню *Project* (Проект) 423
13.3.6. Пункт меню *Diagram* (Диаграмма) 424
13.3.7. Пункт меню *Team* (Команда) 425
13.3.8. Пункт меню *Tools* (Инструменты) 428
13.3.9. Пункт меню *Window* (Окно) 429
13.3.10. Пункт меню *Help* (Справка) 429

Глава 14. Организация работы над проектом в среде Borland Together Designer 430

14.1. Разработка диаграммы вариантов использования в среде Together Designer 431
14.1.1. Создание нового проекта и новой диаграммы вариантов использования 431
14.1.2. Добавление актеров 436
14.1.3. Добавление границы системы 437

14.1.4. Добавление вариантов использования 438
14.1.5. Добавление ассоциаций 439
14.1.6. Добавление зависимостей....441
14.1.7. Добавление текстового файла со сценарием варианта использования 443
14.2. Разработка диаграммы классов в среде Together Designer 444
14.2.1. Добавление классов....446
14.2.2. Добавление атрибутов классов 447
14.2.3. Добавление операций классов....448
14.2.4. Добавление отношений на диаграмму классов 453
14.3. Разработка диаграммы композитной структуры в среде Together Designer....455
14.3.1. Добавление классов и частей 457
14.3.2. Добавление портов и интерфейсов 458
14.3.3. Добавление отношений на диаграмму композитной структуры 460
14.4. Разработка диаграммы последовательности в среде Together Designer....460
14.4.1. Добавление линий жизни....461
14.4.2. Добавление сообщений....462
14.4.3. Добавление комбинированных фрагментов 464
14.5. Разработка диаграммы коммуникации в среде Together Designer 468
14.5.1. Добавление линий жизни....470
14.5.2. Добавление сообщений....471

Глава 15. Завершение разработки проекта в среде Borland Together Designer 474

15.1. Разработка диаграммы деятельности в среде Together Designer....474
15.1.1. Добавление действий и деятельностей....476
15.1.2. Добавление потока управления....477
15.2. Разработка диаграммы конечного автомата в среде Together Designer....479
15.2.1. Добавление состояний 481
15.2.2. Добавление переходов 482
15.3. Разработка диаграммы компонентов в среде Together Designer 484
15.3.1. Добавление компонентов....485
15.3.2. Добавление отношений на диаграмму компонентов 488
15.4. Разработка диаграммы развертывания в среде Together Designer 488
15.4.1. Добавление узлов, сред выполнения и компонентов....490
15.4.2. Добавление отношений на диаграмму развертывания 491
15.5. Генерация документации и программного кода в среде Together Designer....493

Заключение **497**

ПРИЛОЖЕНИЯ **499**

Приложение 1. Язык объектных ограничений OCL **501**

П1.1. Выражения языка OCL 503
П1.2. Основные типы значений и операций в языке OCL 505
П1.3. Операции над отдельными типами значений 507
П1.3.1. Операции с действительными числами 507
П1.3.2. Операции с целыми числами 509
П1.3.3. Операции со строками 511
П1.3.4. Операции с булевыми выражениями 512
П1.3.5. Операция *@pre* для указания предшествующих элементов 514
П1.4. Допустимые выражения в языке OCL 514
П1.5. Неопределенное выражение 515
П1.6. Коллекции значений в языке OCL 515
П1.7. Операции над коллекциями значений 516
П1.7.1. Операция выбора *select* 516
П1.7.2. Операция исключения *reject* 517
П1.7.3. Операция формирования коллекции *collect* 517
П1.7.4. Операция "для всех" *forAll* 518
П1.7.5. Операция "существует" *exists* 518
П1.7.6. Другие операции над коллекциями значений 519
П1.8. Некоторые операции с множествами, последовательностями и комплектами 520
П1.9. Операции преобразования типов 521
П1.10. Примеры записи выражений языка OCL 522
П1.10.1. Определение значения переменной 522
П1.10.2. Определение возраста сотрудника 522
П1.10.3. Определение кратности значений 522
П1.10.4. Определение коллекции инвариантов 522

Приложение 2. Глоссарий **524**

Литература **547**

Предметный указатель **551**

Предисловие

Книга посвящена последней версии унифицированного языка моделирования или, сокращенно, языка UML (Unified Modeling Language), который предназначен для описания, визуализации и документирования объектно-ориентированных систем и бизнес-процессов с ориентацией на их последующую реализацию в виде программного обеспечения.

Хотя в настоящее время отечественные разработчики применяют на практике несколько нотаций визуального моделирования, именно нотация UML все более активно используется системными аналитиками, архитекторами и руководителями проектов для разработки графических моделей при выполнении программных проектов. При этом последние версии интегрированных сред разработки приложений MS Visual® Studio .NET® и Borland® Delphi® Studio не только поддерживают нотацию языка UML в качестве базовой технологии моделирования программных систем, но и позволяют получить исполнимые модули программ на основе разработанной графической модели.

Возрастающая сложность прикладных программ и высокая стоимость их разработки и сопровождения практически не оставляют выбора в вопросе: стоит ли изучать язык UML? В разработке современных корпоративных информационных систем принимают участие десятки, а то и сотни различных специалистов, для которых построение предварительной модели системы до начала написания соответствующего программного кода становится настоятельной необходимостью. Разработка программных систем на заказ также приводит к необходимости поддержания единого стиля для различных версий программ при их постоянной доработке и модификации.

Основное требование к модели программной системы — модель должна быть понятна как заказчику, так и всем специалистам проектной группы, включая и программистов. Оказалось, что разработка соответствующего языка моделирования или нотации является непростым делом. Потребовалось

несколько лет, прежде чем усилия группы специалистов ведущих фирм-производителей программного и аппаратного обеспечения привели к появлению языка UML 2.0.

Какими бы свойствами ни обладал новый язык, важной особенностью для его жизнеспособности является реализация и поддержка соответствующей нотации в коммерческих программных продуктах. В последние годы практически наблюдается все более активный интерес к языку UML, о чем свидетельствуют десятки коммерческих программных инструментариев, предназначенных для автоматизации разработки программного обеспечения на основе построения предварительной объектно-ориентированной модели предметной области. Одним из первых инструментальных средств, в котором была реализована разработка моделей в нотации UML 2.0, является программа Borland® Together® Designer.

Однако наибольшее впечатление производит возможность синхронизации кода и моделей в нотации UML при интеграции этой программы с инструментальными средами программирования, такими как Borland® Together® Developer for JBuilder® и Borland® Together® Developer for MS .NET®. И, наконец, нельзя не упомянуть о среде Borland® Delphi® Studio 2006, в которой реализованы в полном объеме возможности разработки приложений, управляемых моделями в нотации UML. Изучив материал книги, читатели без особого труда смогут освоить редактор моделей в Borland® Delphi® Studio и самостоятельно разрабатывать управляемые моделями приложения в этой среде.

Язык UML 2.0 находит эксклюзивное применение во многих современных информационных технологиях, среди которых следует отметить концепцию управляемой моделями архитектуры (Model Driven Architecture, MDA), в рамках которой поддерживается управляемый моделями подход для разработки программного обеспечения. При разработке платформенно-независимых моделей (PIM), которые концентрируют внимание на общей архитектуре системы, используется только нотация языка UML 2.0.

Следует отметить еще одну область, в которой язык UML 2.0 является единственным средством визуального моделирования и документирования. Это так называемые паттерны (patterns) проектирования, которые концентрируют в себе положительный архитектурный опыт разработки программных приложений. Уже сейчас в некоторых интегрированных средах программирования можно разработать нетривиальное приложение, не написав ни одной строчки программного кода. И эта тенденция, как это не может показаться парадоксальным, со временем будет только усиливаться.

Структура книги

Материал книги соответствует последней версии языка UML 2.0. В основу книги положены две основные идеи. С одной стороны, рассмотреть все базовые конструкции языка UML 2.0, без понимания которых вряд ли возможно адекватно и безошибочно использовать богатейший потенциал возможностей языка UML 2.0. С другой стороны, донести до читателя основы методологии объектно-ориентированного анализа и проектирования, которая необходима для самостоятельной разработки концептуальных, логических и физических моделей программных систем и бизнес-процессов.

Материал книги делится на три части. *Часть I* знакомит с основными понятиями объектно-ориентированного анализа и проектирования, а также с историей развития языка UML 2.0, что представляется совершенно необходимым для правильного понимания назначения и возможностей языка UML 2.0. Поскольку UML 2.0 не является формальным языком с фиксированным синтаксисом, описание языка рассматривается как некоторая открытая модель с определенными базовыми семантическими конструкциями и неформальными правилами их расширения.

Часть II является центральной в книге и содержит описание назначения элементов всех канонических диаграмм языка UML 2.0, которые являются основой построения концептуальных, логических и физических моделей. В отдельных главах этой части последовательно рассматриваются все 13 канонических диаграмм языка UML 2.0:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграмма композитной структуры;
- диаграмма пакетов;
- диаграмма объектов;
- диаграмма последовательности;
- диаграмма деятельности;
- диаграмма коммуникации;
- диаграмма обзора взаимодействия;
- временная диаграмма;
- диаграмма конечного автомата;
- диаграмма компонентов;
- диаграмма развертывания.

Для каждой из диаграмм описываются все базовые элементы графической нотации, даются определения всех ключевых терминов, рассматривается их семантика и особенности графического изображения на различных диаграммах, а также приводятся примеры диаграмм, иллюстрирующие соответствующие понятия языка UML 2.0. Материал этих двух частей книги абсолютно не зависит от средств реализации языка UML 2.0 и может быть использован при разработке моделей с использованием любого CASE-средства, например, IBM Rational® Software Architect (IBM Corp) или Enterprise Architect (Sparx Systems).

Часть III содержит описание особенностей реализации языка UML 2.0 в уже упомянутом CASE-инструментарии — Borland® Together® Designer. В этой части также представлен сквозной пример — последовательная разработка всех диаграмм системы управления банкоматом. После его изучения читатели без особого труда смогут разработать соответствующие диаграммы и в любых других CASE-средствах, поддерживающих нотацию языка UML 2.0.

Рекомендации по изучению языка UML

Современные тенденции развития индустрии создания программного обеспечения складываются таким образом, что именно язык UML 2.0 де-факто оказывается общепризнанным стандартом в области разработки моделей систем и процессов с его последующей реализацией в соответствующих инструментальных средствах.

В то же время следует отметить одну важную особенность современной программной инженерии — это тенденция специализации в области разработки программ. Речь идет о появлении таких специалистов, как системный аналитик, менеджер проекта, бизнес-аналитик и архитектор системы, которые, наряду со знанием языков программирования, должны владеть методологией объектно-ориентированного анализа и моделирования предметной области. Для системного инженера и интегратора также важно разбираться в возможностях реализации конкретных проектов и использовать общепринятую систему обозначений для решения своих задач. Наконец, корпоративные программисты, занятые в масштабных проектах, должны четко понимать функциональные аспекты будущей программной системы. Для всех этих категорий специалистов и предназначена данная книга, поскольку язык UML 2.0 позволяет организовать эффективное и понятное для всех общение.

Уже сейчас элементы нотации языка UML 2.0 начинают использоваться в учебных программах для обучения студентов технических специальностей. Во всяком случае, многие преподаватели давно осознали ограниченность

существующих отечественных стандартов на разработку программной документации, которые не отражают современных тенденций развития программной инженерии. Как студенты, так и преподаватели найдут в книге интересный материал для размышления, который позволит понять целый ряд особенностей и перспектив профессионального образования в области современных информационных технологий.

Для понимания основных конструкций языка UML 2.0 достаточно общей эрудиции. Читатели, которые ставят перед собой такую цель, могут бегло просмотреть материал первой части и сразу приступить к изучению отдельных диаграмм. Однако для более углубленного его изучения полезно знакомство с одним из языков программирования. Для творческого овладения методологией объектно-ориентированного анализа и проектирования необходима, как представляется, определенная строгость мышления и знание некоторых общих понятий прикладного системного анализа. Соответствующий материал приводится в первой части книги и далее используется по мере изложения элементов конкретных диаграмм.

Для лучшего восприятия изложенного в книге материала используются следующие символы.

Этот символ указывает на определение отдельных терминов языка UML 2.0 или ООАП. При этом приводится оригинальное имя на английском и его перевод.

Этот символ обращает внимание на важную особенность или правило языка UML 2.0. Невыполнение или игнорирование этого правила или условия может послужить источником ошибок в представлении графических диаграмм языка UML 2.0.

Этот символ указывает на примечание, которое уточняет или дополняет основной материал. Приводимый здесь текст не оказывает влияния на нотацию и семантику языка UML 2.0.

Благодарности

Автор искренне признателен всем сотрудникам УКЦ Interface Ltd (**www.interface.ru**) за совместную творческую работу. В понимании отдельных вопросов существенную помощь оказали неформальные консультации с сотрудниками компании Borland, которым автор выражает свою признательность. Написание современной книги немыслимо без использования ресурсов

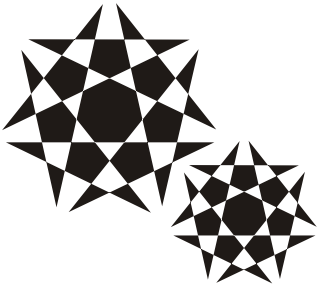
Интернета. В этой связи хотелось бы выразить неизменную признательность директору Междисциплинарного Центра СПбГУ (**www.icape.nw.ru**) профессору Н. В. Борисову за предоставленную возможность электронной коммуникации.

Постскриптум

В основу написания книги положена оригинальная спецификация языка UML 2.0, которая за время работы над книгой дважды меняла свою редакцию. При этом изменился, а вернее, существенно увеличился состав ее авторов. Это не могло не отразиться на содержании спецификации. Именно по этой причине автор позволил себе в отдельных случаях сделать некоторые уточнения, способствующие, по его мнению, лучшему пониманию конструкций языка UML 2.0.

Последняя версия языка UML 2.0 во многих аспектах серьезно отличается от предыдущих версий. Не имея представления о многочисленных нюансах этих отличий, работа над книгой вначале представлялась автору несложной и связанной, главным образом, с дополнением и уточнением семантики отдельных понятий. На деле все оказалось иным, поскольку версия языка UML 2.0 в действительности оказалась и сложной, и масштабной. Такой же оказалась и работа над книгой.

Завершая работу, автор искренне надеется, что читатели оценят этот труд, поскольку книги вполне достаточно для самостоятельного овладения всеми конструкциями языка UML 2.0. Если же ее окажется недостаточно, то всех читателей автор будет рад видеть на своих курсах по соответствующей тематике в УКЦ Interface Ltd. В любом случае автор будет признателен за все отзывы и конструктивные предложения, связанные с содержанием книги и проблематикой моделирования в контексте языка UML 2.0, которые можно отправлять по адресу издательства "БХВ-Петербург" **mail@bhv.ru**.

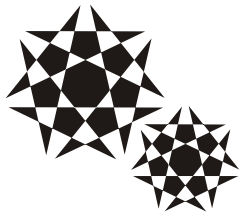


Часть I

Основы объектно-ориентированного анализа и проектирования информационных систем

Данная часть книги посвящена изложению концептуальных основ новой методологии разработки программных приложений — объектно-ориентированному анализу и проектированию. Хотя первые графические системы обозначений для описания структуры программ возникли еще в процедурном программировании, именно в рамках этой новой методологии получил развитие унифицированный язык моделирования — язык UML. Для понимания основ этого языка необходимо не только владеть терминологией объектно-ориентированного программирования, но и иметь общее представление о возможностях визуального моделирования в контексте концепции разработки архитектур, управляемых моделями. В этой части книги излагаются базовые понятия моделирования систем и программных приложений, а также приводятся сведения по наиболее распространенным графическим нотациям. Описываются основные элементы языка UML 2.0 с их краткой характеристикой. В последующем эти элементы рассматриваются более подробно при изучении отдельных видов канонических диаграмм в *части II*.

Глава 1



Базовые принципы и понятия технологии разработки объектно-ориентированных информационных систем

Интенсивное развитие компьютерных и информационных технологий в последние годы тесно связано с использованием все более сложных концепций, которые аккумулируют в себе самые последние достижения в сфере науки и техники. Внедрение в практику современного бизнеса корпоративных информационных систем, адаптация их функционала к нуждам конкретных компаний, выполнение масштабных проектов по разработке программных систем и аппаратно-программных комплексов происходит на фоне постоянного усложнения как физических и программных компонентов, так и лежащих в их основе концепций и идей. Целый ряд конкурентных преимуществ в современном бизнесе тесно связан с использованием последних достижений в области информационных технологий.

Одновременно с увеличением стоимости информационных проектов заметно возросли риски успешного их выполнения. Анализ отрицательных результатов свидетельствует о вполне сложившихся тенденциях, среди которых, наряду с низкой квалификацией персонала рабочих групп и менеджеров проектов, одно из главных мест занимает отсутствие адекватных моделей и архитектур разрабатываемых программных приложений. Все эти особенности приводят к настоятельной необходимости моделирования структуры и процесса функционирования программных систем до начала написания соответствующего кода. При этом непременным условием успешного завершения проекта становится построение предварительной визуальной модели программной системы.

1.1. Основные понятия моделирования систем и программных приложений

Модель (model) — абстракция произвольной системы или объекта, рассматриваемая с определенной точки зрения и представленная на некотором языке или в графической форме.

Модель является абстракцией физической или ментальной системы в контексте некоторой цели. Эта цель определяет, что должно быть включено в модель, а что является неинтересным и может быть проигнорировано без ущерба для ее понимания. Другими словами, модель должна полностью описывать только те аспекты системы, которые являются релевантными конкретной цели моделирования на подходящем уровне детализации.

Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. При этом процесс построения и последующего применения моделей для получения информации о системе-оригинале получил название *моделирование*.

Модель состоит из множества элементов, которые совместно описывают моделируемую систему. При этом модель сложной системы может быть представлена в некоторой иерархической форме и в виде различных представлений. Каждое такое представление специфицирует рассматриваемую систему с определенной точки зрения в интересах определенной категории заинтересованных лиц, например разработчиков, пользователей или клиентов этой системы, и на определенном уровне абстракции.

Представление (view) — проекция модели, которая рассматривается с определенной точки зрения и учитывает только существенные аспекты модели.

Модель может считаться законченной, если она представляет физическую систему в целом, хотя в ней могут быть представлены только те аспекты, которые являются релевантными ее цели, т. е. на заданном уровне абстракции и с заданной точки зрения. Элементы модели могут быть организованы в некоторую иерархию, в которой наиболее верхнее представление или подсистема выступает в качестве границы рассматриваемой системы.

Важнейшими характеристиками любой системы являются ее структура и процесс функционирования. При этом под *структурой системы* понимают

устойчивую во времени совокупность взаимосвязей между ее элементами или компонентами. Именно структура связывает воедино все элементы и препятствует распаду системы на отдельные составляющие ее элементы. Структура системы может отражать самые различные взаимосвязи, в том числе и вложенность элементов одной системы в другую. В этом случае принято называть более мелкую или вложенную систему *подсистемой*, а более крупную — *метасистемой*.

Структуру программных систем на верхнем уровне представления часто называют архитектурой, понимая под этим термином наиболее общие структурные компоненты приложений и взаимосвязи между ними.

Процесс функционирования системы отражает поведение системы во времени и может быть представлен как последовательное изменение ее состояний. Если система изменяет одно свое состояние на другое состояние, то принято говорить, что система переходит из одного состояния в другое. Совокупность признаков или условий изменения состояний системы в этом случае называется переходом. Для системы с дискретными состояниями процесс функционирования может быть представлен в виде последовательности состояний с соответствующими переходами.

Для одной и той же физической системы могут быть разработаны различные модели, которые могут отличаться формой и уровнем представления информации об этой системе. При этом различные модели должны дополнять друг друга и специфицировать разные аспекты системы с различных перспектив (точек зрения). Модели могут уточнять друг друга и декомпозироваться на более детальные представления. Обязательное наличие нескольких моделей для адекватного представления особенно характерно для моделирования сложных систем.

Сложность системы и, соответственно, ее модели может быть рассмотрена с различных точек зрения. Прежде всего, можно выделить сложность структуры системы, которая характеризуется количеством элементов системы и различными типами взаимосвязей между этими элементами. Если количество элементов превышает некоторое пороговое значение, которое, вообще говоря, не является фиксированным, то такая система может быть названа сложной. Например, если программное приложение насчитывает более 100 отдельных форм ввода и вывода информации, то многие программисты сочтут ее сложной. Транспортная система современных мегаполисов также может служить примером сложной системы.

Вторым аспектом сложности является сложность процесса функционирования системы. Это может быть связано как с непредсказуемым характером поведения системы, так и невозможностью формального представления правил преобразования входных воздействий в выходные. В качестве примеров

сложных программных систем можно привести современные операционные системы, которым присущи черты сложности как структуры, так и поведения.

Процесс разработки моделей можно представить как поуровневый спуск от наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня. При этом на каждом из этапов процесса разрабатываемые модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты реализации сложной системы. Общая архитектура моделей и представлений сложной системы схематично изображена на рис. 1.1.

Рис. 1.1. Общая архитектура моделей и представлений сложной системы

Форма представления моделей тесно связана с исходными целями моделирования. В практике разработки программных систем и приложений наибольшее распространение получили визуальные модели, которые используют для представления своих элементов специальную графическую нотацию.

Нотация (notation) — система условных обозначений, специально разработанная для представления элементов модели в графической форме.

Важным достоинством той или иной графической нотации является возможность образного закрепления содержательного смысла отдельных понятий за тем или иным геометрическим символом, что существенно упрощает процесс моделирования и интерпретации соответствующих моделей среди разработчиков. Основное требование к модели программной системы — модель должна быть понятна заказчику и всем специалистам проектной группы, включая бизнес-аналитиков и программистов. Именно для разработки такой нотации потребовались усилия группы специалистов ведущих фирм производителей программного и аппаратного обеспечения, которые привели к появлению языка UML.

Унифицированный язык моделирования (Unified Modeling Language, UML) — язык визуального моделирования, предназначенный для спецификации, визуализации и документирования объектно-ориентированных систем и бизнес-процессов во время их проектирования и разработки.

Следует отметить, что довольно часто встречаются неточные варианты перевода этого термина и неверная интерпретация сокращения UML. Так, например, даже в англо-язычных источниках UML иногда называют универсальным языком, забывая его первоначальное определение.

Использование той или иной нотации, обладающей широким набором графических символов для представления визуальных моделей, вообще говоря, еще не отвечает на вопрос о том, каким образом и в какой последовательности необходимо разрабатывать сами модели. Если ранние нотации изображения схем программных приложений, как правило, содержали рекомендации по методике их построения, то с появлением более сложных нотаций ситуация кардинально изменилась. Главной особенностью развития современных технологий разработки программных систем стало появление различных методологий, которые детально описывают сам процесс разработки независимо от используемых при этом нотаций.

Методология (methodology) — совокупность принципов и методик разработки программных систем и приложений, определяющих процесс построения моделей при выполнении соответствующих проектов.

Современные методологии разработки программных приложений тесно связаны с концепцией автоматизированной разработки программного обеспечения

(Computer Aided Software Engineering, CASE) и соответствующими программными средствами, реализующими эту концепцию.

Хотя появление первых CASE-средств было встречено с некоторой настороженностью, со временем появились как восторженные отзывы об их применении, так и критические оценки их возможностей. Причин для столь противоречивых мнений было несколько. Первая из них заключается в том, что ранние CASE-средства были простой надстройкой над некоторой системой управления базами данных (СУБД). Хотя визуализация процесса разработки концептуальной схемы БД имеет немаловажное значение, она не решает проблем разработки приложений других типов.

CASE-средства (CASE-tools) — программное обеспечение, которое предназначено для разработки визуальных моделей программных приложений и генерации исходного кода на некотором языке программирования или схемы базы данных.

Вторая причина имеет более сложную природу, поскольку связана с графической нотацией, реализованной в том или ином CASE-средстве. При этом попытки предложить универсальную систему обозначений, как для визуального представления концептуальных схем БД, так и для архитектуры программных приложений, закончились безрезультатно. На этом фоне разработка и стандартизация унифицированного языка моделирования UML были восприняты с большим оптимизмом всем сообществом корпоративных программистов.

Современные CASE-средства не только поддерживают одну или несколько графических нотаций, но и могут быть встроены в интегрированные среды разработки приложений, такие как Borland Studio или MS Visual Studio. Дополнительная поддержка технологий синхронизации программного кода и визуальных моделей позволяет не только реализовать двунаправленную разработку приложений, но и существенно сократить сроки выполнения проектов. Взаимосвязь нотации, методологии и CASE-средств при выполнении проектов можно условно изобразить в форме треугольника, при этом выбор конкретных элементов представляет лишь один из возможных вариантов, который и рассматривается в данной книге (рис. 1.2).

Разработка и использование моделей в нотации UML осуществляется в рамках общей методологии объектно-ориентированного анализа и проектирования, которая, в свою очередь, является обобщением и дальнейшим развитием методологии объектно-ориентированного программирования. Далее рассматриваются основные принципы и понятия этой методологии, которые служат концептуальным базисом для понимания всех основных понятий языка UML 2.0.

Рис. 1.2. Взаимосвязь нотации, методологии и CASE-средств

1.2. Методология объектно-ориентированного анализа и проектирования

Методология объектно-ориентированного анализа и проектирования, которая пришла на смену методологии объектно-ориентированного программирования, существенно расширила сферу своих интересов, включив в нее не только процесс написания программного кода, но и разработку моделей, а также анализ архитектур приложений. Наиболее существенным в новой методологии явилось осознание того факта, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы.

Действительно, до того, как начать программирование классов, их свойств и методов, необходимо дать ответы на такие вопросы, как: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами. Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа.

Объектно-ориентированный анализ и проектирование (Object-Oriented Analysis/Design) — методология разработки программных систем, в основу которой положена объектно-ориентированная концепция представления моделей предметной области в форме классов, обладающих структурными свойствами и поведением.

Фундаментальными понятиями методологии ООАП являются собственно понятия класса и объекта, а также ее основные принципы — абстракция, наследование, инкапсуляция и полиморфизм.

Абстракция (abstraction) — характеристика сущности, которая отличает ее от других сущностей.

Абстракция определяет границу представления соответствующего элемента модели и используется для определения фундаментальных понятий ООАП, которыми являются понятия класса и объекта.

Класс (class) — абстракция совокупности реальных объектов, которые имеют общий набор свойств и обладают одинаковым поведением.

Каждый объект в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых свойств или не обладают одинаковым поведением, по определению не могут быть отнесены к одному классу.

Важной особенностью классов является возможность их организации в виде некоторой иерархической структуры, которая по внешнему виду напоминает схему классификации понятий формальной логики. Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории выбирается понятие, имеющее наибольший объем и, соответственно, наименьшее содержание. Это самый высший уровень абстракции для рассматриваемой иерархии. Затем наиболее общее понятие некоторым образом конкретизируется, тем самым уменьшается его объем, но увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии располагается на уровень ниже исходного понятия.

Этот процесс конкретизации понятий может быть продолжен до тех пор, пока на самом нижнем уровне не будут получены понятия, дальнейшая конкретизация которых в данном контексте либо невозможна, либо нецелесообразна. Если в качестве понятий использовать классы, то такая иерархия может служить иллюстрацией принципа наследования ООАП.

Наследование (inheritance) — принцип, в соответствии с которым знание о более общей категории разрешается применять для более частной категории.

Наследование тесно связано с иерархией классов, которая определяет, какие классы следует считать наиболее абстрактными и общими по отношению к другим классам. При этом если некоторый общий или родительский класс (предок) обладает фиксированным набором свойств и поведением, то любой наследуемый от него класс (потомок) должен содержать этот же набор свойств и поведение, а также, возможно, иметь дополнительные, которые будут характеризовать уникальность данного класса-потомка. В этом случае просто говорят, что класс-потомок наследует свойства и поведение класса-предка.

Для иллюстрации принципа наследования можно привести следующий пример. Рассмотрим в качестве общего класс `Автомобиль`. Этот класс определяется как некоторая абстракция свойств и поведения всех реально существующих автомобилей. При этом свойствами класса `Автомобиль` могут быть такие общие свойства, как наличие двигателя, трансмиссии, колес и рулевого управления. Если в качестве класса-потомка рассмотреть класс `Легковой автомобиль`, то все выделенные выше свойства будут присущи и этому классу. Можно сказать, что класс `Легковой автомобиль` наследует свойства родительского класса `Автомобиль`. Однако кроме перечисленных свойств этот класс-потомок будет содержать дополнительные свойства, например, такое как наличие салона с количеством посадочных мест 2—5.

В свою очередь, класс `Легковой автомобиль` может быть классом-предком для других классов, которые вполне могут соответствовать, например, моделям конкретных фирм-производителей. С этой точки зрения можно рассматривать класс `Легковой автомобиль производства ВАЗ`. Поскольку Волжский автомобильный завод выпускает несколько моделей автомобилей, одним из классов-потомков для предыдущего класса может быть конкретная модель автомобиля, например, ВАЗ-2110. Рассмотренный фрагмент классификации автомобилей может быть представлен графически в форме диаграммы классов в нотации UML 2.0 (рис. 1.3).

Следует заметить, что диаграмма классов на рис. 1.3 является неполной в том смысле, что на ней не представлены другие модели автомобилей. В дополнение к этому все классы, кроме классов самого нижнего уровня, являются абстрактными. При этом никакие свойства классов здесь не указаны. Все это должно навести на серьезные размышления по поводу требований к графической нотации, которая должна позволять представлять дополнительную информацию о классах.

В языке UML 2.0 для представления классификаций и наследования классов используется специальное отношение обобщения *(см. главу 3)*.

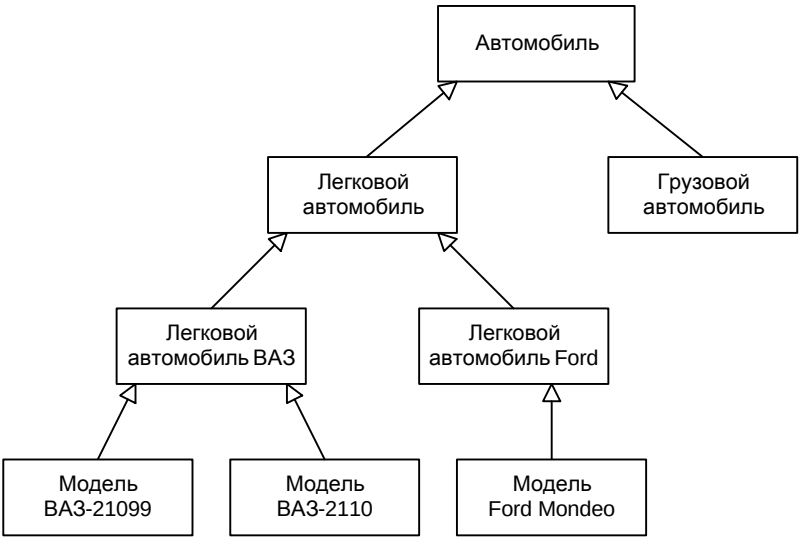


Рис. 1.3. Фрагмент схемы классификации автомобилей в нотации UML 2.0

Следующий принцип ООАП — инкапсуляция.

Инкапсуляция (encapsulation) характеризует сокрытие отдельных деталей внутреннего устройства классов или компонентов от внешних по отношению к нему объектов или пользователей.

Действительно, взаимодействующему с классом клиенту нет необходимости знать, каким образом реализован тот или иной метод класса, услугами которого он решил воспользоваться. Конкретная реализация присущих классу свойств и методов, которые определяют поведение этого класса, является собственным делом данного класса. Более того, отдельные свойства и методы класса вообще могут быть невидимы за пределами этого класса, что является основной идеей введения различных форм видимости для элементов класса.

Если продолжить рассмотрение примера с классом `Легковой автомобиль`, то нетрудно проиллюстрировать инкапсуляцию следующим образом. Основным пользователем, который взаимодействует с объектами этого класса, является водитель. Очевидно, что не каждый водитель в совершенстве знает внутреннее устройство легкового автомобиля. Более того, отдельные детали этого устройства сознательно скрыты в корпусе двигателя или коробке передач. А в случае нарушения работы автомобиля или при возникновении неисправностей в работе его подсистем необходимый ремонт выполняет профессиональный механик.

Инкапсуляция ведет свое происхождение от деления модулей в некоторых языках программирования на две части или секции: интерфейс и реализацию. При этом в интерфейсной секции модуля описываются все объявления функций и процедур, а возможно и типов данных, которые доступны за пределами данного модуля. Другими словами, указанные процедуры и функции являются способами оказания услуг внешним клиентам. В другой секции модуля, называемой реализацией, содержится программный код, который определяет конкретные способы реализации объявленных в интерфейсной части процедур и функций.

Принцип разделения модуля на интерфейс и реализацию отражает суть наших представлений об окружающем мире. В интерфейсной части указывается вся информация, необходимая для взаимодействия с любыми другими объектами и пользователями. Реализация скрывает или маскирует от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. При этом доступ к секции реализации имеет только разработчик (рис. 1.4).

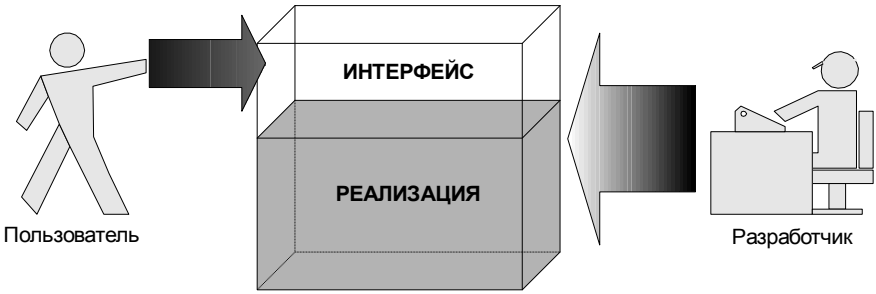


Рис. 1.4. Иллюстрация принципа инкапсуляции для сокрытия внутренних деталей классов

Третьим принципом ООАП является полиморфизм. В общем случае под полиморфизмом (от греч. *poly* — много, *morfos* — форма) понимают свойство некоторых объектов принимать различные внешние формы в зависимости от обстоятельств. Применительно к ООАП используется более частное определение.

Полиморфизм (polymorphism) — свойство одноименных методов или операций выполнять различные действия или обладать различным поведением в зависимости от того, к какому из классов они относятся.

Рассмотрим, например, три объекта соответствующих классов: двигатель автомобиля, электрический свет в комнате и персональный компьютер.

Для каждого из них можно определить операцию `включить`. Однако результат выполнения этой операции будет отличаться для каждого из рассмотренных объектов. Так для двигателя автомобиля вызов метода `включить` означает его запуск. Вызов метода `включить` для комнаты означает простой щелчок выключателя, в результате чего комната наполнится светом. Вызов метода `включить` для персонального компьютера приведет к серии действий по загрузке установленной операционной системы, после чего компьютер будет готов выполнить операции пользователя.

Для операции `включить`, рассматриваемой в том или ином контексте, можно определить дополнительные параметры, такие как время выключения, некоторое условие нахождения объекта в работоспособном состоянии и пр. С этой целью после имени операции указываются скобки, в которых могут быть перечислены эти дополнительные параметры или аргументы. В случае отсутствия аргументов считается, что список параметров пуст. Однако скобки все равно записываются и указывают на тот факт, что соответствующее имя является именем операции или метода, в отличие от свойств или атрибутов класса, которые записываются без скобок.

Полиморфизм в объектно-ориентированных языках связан с *перегрузкой* функций, но не тождествен ей. Важно иметь в виду, что имена методов и свойств тесно связаны с классами, в которых они описаны. Это обстоятельство обеспечивает определенную надежность работы программы, поскольку исключает случайное применение метода для решения несвойственной ему задачи.

Для схематичного представления визуальных моделей в рамках концепции ООАП было предложено большое количество графических нотаций, среди которых наибольшее распространение получили следующие, получившие названия *диаграмм*:

- диаграммы "сущность — связь" (Entity-Relationship Diagrams, ERD);
- диаграммы функционального моделирования (Structured Analysis and Design Technique, SADT);
- диаграммы потоков данных (Data Flow Diagrams, DFD).

Диаграммы "сущность — связь" (ERD) предназначены для графического представления моделей данных разрабатываемой программной системы и предлагают некоторый набор стандартных обозначений для определения данных и отношений между ними. С помощью этого вида диаграмм можно описать отдельные компоненты концептуальной модели данных и совокупность взаимосвязей между ними, имеющих существенное значение для разрабатываемой системы.

В рамках диаграмм функционального моделирования было разработано несколько графических нотаций моделирования, которые получили следующие названия:

- нотация IDEF0 — для документирования процессов производства и отображения информации об использовании ресурсов на каждом из этапов проектирования систем;
- нотация IDEF1 — для документирования информации о производственном окружении систем;
- нотация IDEF2 — для документирования поведения системы во времени;
- нотация IDEF3 — специально для моделирования бизнес-процессов.

Процесс моделирования в нотации IDEF представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели системы какой-либо предметной области. Функциональная модель IDEF отображает структуру процессов функционирования системы и ее отдельных подсистем, т. е. выполняемые ими действия и связи между этими действиями. Для этой цели строятся специальные модели, которые позволяют в наглядной форме представить последовательность выполнения определенных действий. Так, например, исходными строительными блоками модели в нотации IDEF0 процесса являются деятельность (activity) и стрелки (arrows).

Основой визуального моделирования информационных систем с помощью диаграмм потоков данных является специальная технология построения диаграмм потоков данных DFD. В разработке методологии DFD приняли участие многие аналитики, среди которых следует отметить Э. Йордона. Он является автором одной из первых графических нотаций DFD.

Основной недостаток рассмотренных нотаций связан с отсутствием графических средств для объектно-ориентированного представления моделей сложных систем, а также для представления нетривиальных алгоритмов обработки данных. Поскольку на рассмотренных типах диаграмм не указываются характеристики времени выполнения отдельных процессов и передачи данных между процессами, то модели систем, реализующих синхронную обработку данных, не могут быть адекватно представлены в этих нотациях.

Хотя указанные нотации по-прежнему находят практическое применение, однако отмеченные недостатки существенно ограничивают возможность их широкого использования в ООАП. Именно эти особенности послужили исходной предпосылкой появления унифицированного языка моделирования и других современных концепций разработки программных систем, среди которых следует отметить концепцию архитектур, управляемых моделями.

1.3. Концепция разработки архитектур, управляемых моделями

Дополнительный импульс развитию языка UML 2.0 в последние годы придает появление новой концепции разработки архитектур программных приложений, управляемых моделями.

Управляемая моделями архитектура (Model Driven Architecture, MDA) — концепция проектирования архитектур многоплатформенных приложений, в рамках которой поддерживается управляемый моделями подход для разработки программного обеспечения.

Одна из первых спецификаций концепции MDA появилась в 2001 г. в рамках консорциума OMG (Object Management Group), который также является официальным разработчиком спецификаций языка UML. Одним из ключевых элементов концепции MDA является унифицированный язык моделирования, который используется на практике при построении моделей архитектур программных приложений.

Необходимость решения проблем интеграции корпоративных информационных систем, которые в последние годы стали особенно актуальными, привела к появлению сложных технологий разработки и совместного использования разнообразных программных компонентов. Хотя унификация информационных интерфейсов компонентов с помощью языков типа XML и IDL частично решает задачу интеграции приложений, но не обеспечивает решения проблемы повторного использования разработанных компонентов при переходе на другую платформу или технологию. Именно это послужило основой более глубокой архитектурной интеграции посредством создания платформенно-независимых моделей.

Платформенно-независимая модель (Platform Independent Model, PIM) — такое представление системы, которое концентрирует внимание на общей архитектуре системы и скрывает детали, необходимые для ее реализации на отдельной платформе.

Платформенно-независимая модель (PIM-модель) представляет только часть полной спецификации программной системы, которая не изменяется при переходе или замене одной платформы на другую. Именно по этой причине для графического изображения PIM-модели необходимо использовать общецелевой язык моделирования, который позволяет абстрагироваться от специфики платформы или предметной области, в которой система будет использоваться.

Согласно концепции MDA, разработка информационной системы должна начинаться с создания PIM-модели, которая определяет состав, структуру и поведение будущего программного приложения. PIM-модель представляет совокупность архитектурных элементов проектируемой системы и связей между ними на достаточно абстрактном уровне, т. е. без привязки к конкретным языкам программирования, технологиям реализации компонентов и операционным системам. Таким образом, PIM-модель представляет общую архитектуру проектируемой информационной системы и соответствующую предметную область в форме абстрактного описания программных объектов, их свойств, методов и отношений между ними.

Тем не менее все объекты PIM-модели должны быть описаны как можно детальнее. Однако такое детальное описание разрабатывается не на языке программирования, который может быть жестко привязан к платформе. Как показала практика, для этих целей лучше всего подходит язык UML, который обладает свойством независимости от платформ разработки и языков программирования.

На последующих этапах разработки информационных систем необходимо адаптировать PIM-модели к конкретным платформам разработки и технологиям реализации программных компонентов. В результате этого может быть создана одна или несколько платформенно-зависимых моделей.

Платформенно-зависимая модель (Platform Specific Model, PSM) — такое представление системы, которое специфицирует все особенности, необходимые для ее реализации на конкретной платформе.

В рамках концепции MDA платформенно-зависимые модели (PSM-модели) являются отображением абстрактной PIM-модели в конкретную программную среду. При этом под программной средой (или платформой) понимаются конкретные технологии создания корпоративных приложений, такие как MS .NET, Sun One, J2EE, CORBA, а также соответствующие языки и среды программирования. Важнейшей особенностью концепции MDA является спецификация механизмов, обеспечивающих взаимосвязь между различными PSM-моделями, а также правил преобразования PIM-моделей в PSM-модели. Именно это позволит, по замыслу разработчиков MDA, обеспечить многоплатформенность и интеграцию компонентов проектируемых информационных систем.

Процесс разработки программных приложений с использованием концепции MDA включает в себя несколько этапов. На начальных этапах выполняется анализ предметной области и спецификация требований к проектируемой

информационной системе. Далее разрабатывается детальная PIM-модель, которая полностью абстрагируется от особенностей конкретных платформ реализации. После ее разработки создается одна или несколько PSM-моделей, которые, в свою очередь, служат основой для создания или генерации программного кода приложения. После этого приступают к компиляции, сборке и тестированию приложения, по окончании чего выполняется развертывание компонентов приложения. Рассмотренный процесс разработки с использованием концепции MDA упрощенно может быть представлен в виде диаграммы деятельности в нотации UML следующим образом (рис. 1.5).

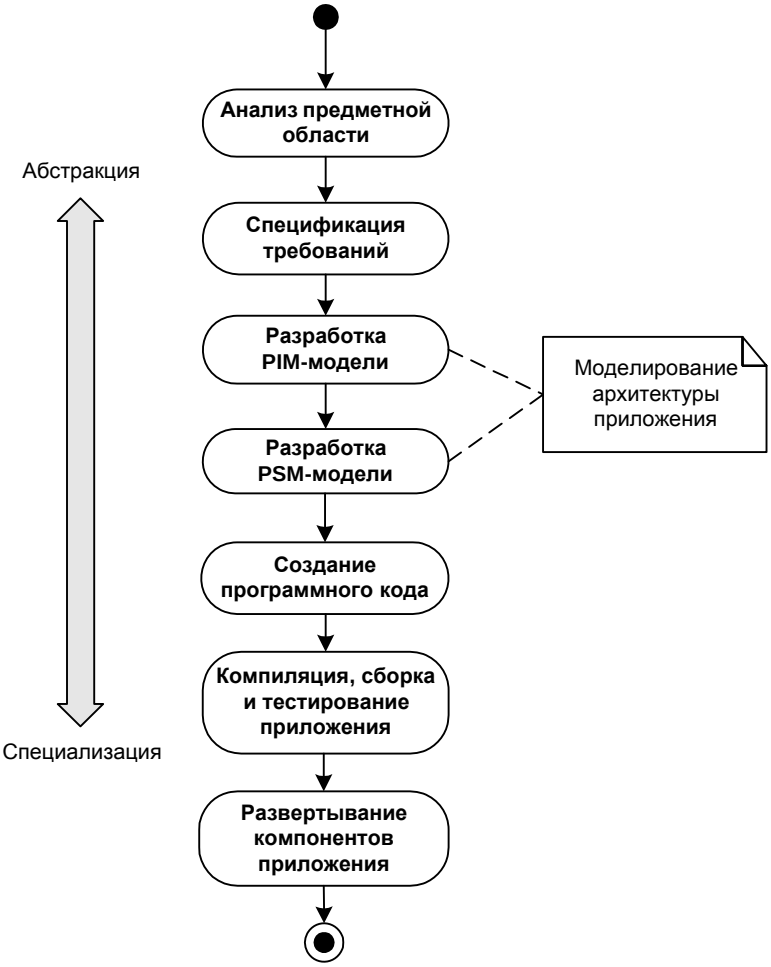


Рис. 1.5. Процесс разработки программных приложений с использованием концепции MDA

Процесс преобразования PIM-модели в конкретные PSM-модели неоднозначен по своей природе и определяется целями создаваемого приложения. Один из возможных вариантов такого преобразования может иметь вид, представленный на рис. 1.6.

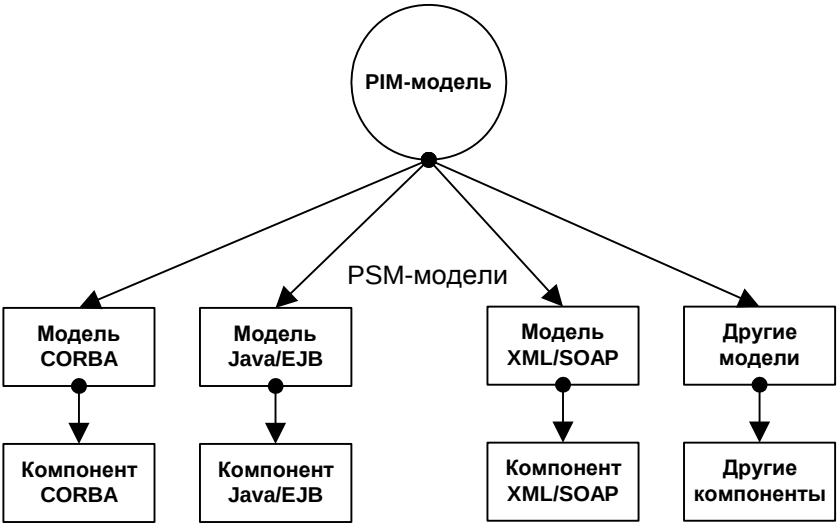


Рис. 1.6. Возможные преобразования PIM-модели в конкретные PSM-модели

Таким образом, согласно концепции MDA, основное внимание разработчиков программных приложений переносится с этапов программирования на этапы создания PIM-модели и ее корректного преобразования в PSM-модели. Эта работа требует хорошего знания графической нотации, которая используется для создания таких моделей, а именно — языка UML. При этом язык UML является не только основным инструментом MDA при создании моделей программных приложений, но и средством документирования всего процесса разработки.

1.4. Основные этапы развития UML 2.0

Отдельные нотации для графического представления объектно-ориентированных моделей стали появляться в период между серединой 70-х и концом 80-х гг. XX столетия, когда различные исследователи и программисты предлагали свои подходы для реализации методологии ООАП. В период между 1989—1994 гг. общее количество наиболее известных нотаций моделирования превысило 50. Многие разработчики испытывали серьезные затруднения при

выборе нотации для своих моделей, поскольку ни одна из них не удовлетворяла всем требованиям, предъявляемым к построению моделей сложных систем. Принятие отдельных методик и графических нотаций в качестве стандартов (IDEF0, IDEF1X) не смогло изменить сложившуюся ситуацию непримиримой конкуренции между ними в начале 90-х годов, которая тогда получила название "войны методов".

К середине 1990-х некоторые из методов были существенно улучшены и приобрели самостоятельное значение при решении отдельных задач в рамках методологии ООАП. Наиболее известными в этот период становятся:

- метод Гради Буча (Grady Booch), получивший условное название Booch или Booch'91, Booch Lite (позже — Booch'93);
- метод Джима Румбаха (Jim Rumbaugh), получивший название Object Modeling Technique — OMT (позже — OMT-2);
- метод Айвара Джекобсона (Ivar Jacobson), получивший название Object-Oriented Software Engineering — OOSE.

Каждый из этих методов был ориентирован на поддержку отдельных этапов ООАП. Например, метод OOSE содержал средства представления вариантов использования, которые имеют существенное значение на этапе анализа требований в процессе проектирования бизнес-приложений. Метод OMT-2 наиболее подходил для анализа процессов обработки данных в информационных системах. Метод Booch'93 нашел наибольшее применение на этапах проектирования и разработки различных программных систем.

История развития языка UML берет начало с октября 1994 г., когда Гради Буч и Джим Румбах из Rational Software Corporation, которая была основана Полом Леви (Paul Levy) и Майком Девлином (Mike Devlin) еще в 1981 г., начали работу по унификации методов Booch и OMT. Хотя сами по себе эти методы были достаточно популярны, совместная работа была направлена на изучение всех известных объектно-ориентированных методов с целью объединения их достоинств. При этом Г. Буч и Дж. Румбах сосредоточили усилия на полной унификации результатов своей работы. Проект так называемого *унифицированного метода* (Unified Method) версии 0.8 был подготовлен и опубликован в октябре 1995 г. Осенью того же года к ним присоединился А. Джекобсон, главный технолог из компании Objectory AB (Швеция), с целью интеграции своего метода OOSE с двумя предыдущими.

Вначале авторы методов Booch, OMT и OOSE предполагали разработать унифицированный язык моделирования только для этих трех методик. С одной стороны, каждый из этих методов был проверен на практике и показал свою конструктивность при решении отдельных задач ООАП. Это давало

основание для дальнейшей их модификации на основе устранения имеющегося несоответствия отдельных понятий и обозначений. С другой стороны, унификация семантики и нотации должна была обеспечить некоторую стабильность на рынке объектно-ориентированных CASE-средств, которая необходима для успешного продвижения соответствующих программных инструментариев. Наконец, совместная работа давала надежду на существенное улучшение всех трех методов.

Начиная работу по унификации своих методов, Г. Буч, Дж. Румбах и А. Джекобсон сформулировали следующие требования к языку моделирования. Он должен:

- позволять моделировать не только программное обеспечение, но и более широкие классы систем и бизнес-приложений, с использованием объектно-ориентированных понятий;
- явным образом обеспечивать взаимосвязь между базовыми понятиями моделей концептуального, логического и физического уровней;
- обеспечивать масштабируемость моделей, что является важной особенностью сложных многоцелевых систем;
- быть понятным аналитикам, программистам и другим специалистам, входящим в состав проектных команд;
- поддерживаться специальными инструментальными средствами, реализованными на различных компьютерных платформах.

Разработка системы обозначений или нотации для ООАП оказалась непохожей на разработку нового языка программирования. Во-первых, необходимо было решить две проблемы:

- Должна ли данная нотация включать в себя спецификацию требований?
- Следует ли расширять данную нотацию до уровня языка визуального программирования?

Во-вторых, было необходимо найти удачный баланс между выразительностью и простотой языка. С одной стороны, слишком простая нотация ограничивает круг потенциальных проблем, которые могут быть решены с помощью соответствующей системы обозначений. С другой стороны, слишком сложная нотация создает дополнительные трудности для ее изучения и применения аналитиками и программистами. В случае унификации существующих методов необходимо учитывать интересы специалистов, которые уже имеют опыт работы с ними, поскольку внесение серьезных изменений в новую нотацию может повлечь за собой непонимание или неприятие ее пользователями прежних методик. Чтобы исключить неявное сопротивление со стороны

отдельных специалистов, необходимо учитывать интересы самого широкого круга пользователей. Последующая работа над языком UML должна была учесть все эти требования.

В этот период поддержка разработки языка UML становится одной из целей консорциума OMG (Object Management Group). Хотя консорциум OMG был образован еще в 1989 г. с целью разработки предложений по стандартизации объектных и компонентных технологий CORBA, язык UML приобрел статус второго стратегического направления в работе OMG. Именно в рамках OMG создается команда разработчиков под руководством Ричарда Соли, которая обеспечила всю дальнейшую работу по унификации и стандартизации языка UML. В июне 1995 г. OMG организовала совещание по методологиям ООАП для всех крупных специалистов и представителей входящих в этот консорциум компаний, на котором впервые в международном масштабе была признана целесообразность поиска индустриальных стандартов в области языков моделирования под эгидой OMG.

Усилия Г. Буча, Дж. Румбаха и А. Джекобсона привели к появлению первых документов, содержащих описание собственно языка UML версии 0.9 (июнь 1996 г.) и версии 0.91 (октябрь 1996 г.). Имевшие статус запроса предложений RTP (Request For Proposals), эти документы послужили своеобразным катализатором для широкого обсуждения языка UML различными категориями специалистов. Первые отзывы и реакция на язык UML указывали на необходимость его дополнения отдельными понятиями и конструкциями.

В это же время стало ясно, что некоторые компании и организации видят в языке UML линию стратегических интересов для своего бизнеса. Компания Rational Software вместе с несколькими организациями, изъявившими желание выделить ресурсы для разработки строгого определения версии 1.0 языка UML, учредила консорциум партнеров UML, в который первоначально вошли такие компании, как Digital Equipment Corp., HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI и Unisys. Эти компании обеспечили поддержку последующей работы по более точному и строгому определению нотации, что привело к появлению версии 1.0 языка UML. В январе 1997 г. был опубликован документ с описанием языка UML 1.0, как начальный вариант ответа на запрос предложений RTP. Эта версия языка моделирования была достаточно хорошо определена, обеспечивала требуемую выразительность и мощность и предполагала решение широкого класса задач.

Возможности инструментальных CASE-средств и диапазон их практического применения в большой степени зависят от удачного определения семантики и нотации соответствующего языка моделирования. Специфика языка UML

заключается в том, что он определяет семантическую метамодель, а не модель конкретного интерфейса и способы представления или реализации компонентов.

Из более чем 800 компаний и организаций, входящих в настоящее время в состав консорциума OMG, особую роль на всех этапах разработки языка UML играла Rational Software Corporation. Эта компания разработала и выпустила в продажу одно из первых инструментальных CASE-средств Rational Rose 98, в котором была реализована нотация различных диаграмм языка UML 1.0.

В феврале 2003 г. компания Rational Software Corporation была приобретена всемирно известной IBM, и с этого момента она имеет официальное название — IBM Rational Software. Соответствующим образом добавилась приставка ко всей линейке ее программных продуктов. После этого приобретения технологии Rational стали неотъемлемой частью портфеля программного обеспечения IBM. "Rational является неотъемлемым элементом нашей недавно объявленной инициативы в области электронного бизнеса по требованию, — утверждает Стив Миллз (Steve Mills), вице-президент IBM и руководитель подразделения IBM Software. — Законченная, открытая программная платформа Rational позволит повысить скорость, качество и прогнозируемость проектов в области программного обеспечения. Продукты Rational станут великолепным дополнением к имеющимся у нас четырем семействам ПО — WebSphere, DB2, Lotus и Tivoli".

В начале 1997 г. целый ряд других компаний, среди которых были IBM, ObjecTime, Platinum Technology и некоторые другие, представили на рассмотрение OMG собственные ответы на запрос предложений RFP. В дальнейшем эти компании присоединились к партнерам UML, предлагая включить в язык UML некоторые свои идеи. В результате совместной работы с партнерами UML была предложена пересмотренная версия 1.1 языка UML. Основное внимание при разработке языка UML 1.1 было уделено достижению большей ясности семантики языка по сравнению с UML 1.0, а также учету предложений новых партнеров. Эта версия языка была представлена на рассмотрение OMG, одобрена и принята в качестве стандарта OMG в ноябре 1997 г.

Компании ObjectTime и Rational Software Inc., начиная с 1997 г., приступили к совместной работе над унификацией своих перспективных технологий. После периода стратегической кооперации, в ходе которой ObjectTime являлась эксклюзивным поставщиком средств моделирования и технологии автоматической генерации кода компании Rational, предназначенных для систем реального времени (а компания Rational имела права на распространение ObjectTime Developer по всему миру), компания Rational Software объявила о приобретении компании ObjectTime в декабре 1999 г.

В результате понятие архитектуры, впервые появившееся в методологии ROOM, теперь нашло свое применение в расширениях реального времени для UML. Язык UML вместе с расширениями реального времени становится, вероятно, наиболее широко распространенным и коммерчески успешным языком описания архитектур программных приложений. Он позволяет моделировать структуру системы, идентифицируя представляющие интерес ее элементы и отношения между ними. В классе систем реального времени, описанных в рамках методологии ROOM (сложные, управляемые событиями и потенциально распределенные системы, похожие на те, которые используются в телекоммуникационных, авиакосмических и оборонных системах), в фокусе расширений реального времени оказались два типа (из восьми) канонических диаграмм UML: диаграммы классов и диаграммы кооперации.

В течение всех последующих лет все вопросы дальнейшей разработки языка UML были сконцентрированы в рамках консорциума OMG. Соответствующая группа специалистов обеспечивает публикацию материалов, содержащих спецификации текущих версий языка UML и запросов предложений RFP по его дальнейшей стандартизации. В марте 1999 г. консорциум OMG опубликовал описание языка UML (alpha R5). Однако следующей версией языка UML стала версия 1.4, специфицированная в сентябре 2001 г., после выхода которой началась работа над следующей версией, которая была анонсирована как UML 2.0. Видимо, именно с этого момента стал проявляться интерес к языку UML 2.0 со стороны прессы, которая, публикуя интервью с "Three amigos", неоднократно указывала на важные нововведения в этой перспективной версии.

Однако, начиная с 2001 г., обещанные сроки принятия и публикации спецификации на версию 2.0 языка UML в силу различных обстоятельств постоянно откладывались. Возможно, для многих совершенной неожиданностью оказалось появление в марте 2003 г. документации на версию 1.5 языка UML, которая была описана в соответствующем документе 03.03.01 — "OMG Unified Modeling Language Specification". Именно эта версия до середины 2005 г. формально являлась текущей официально принятой версией языка UML.

В марте 2003 г. появилась первая спецификация на проект языка UML 2.0, которая была описана в документе ptc/03-07-06. С этого момента вплоть до 2005 г. за этой версией языка прочно утвердилась приставка "проект". Однако работа над языком UML 2.0 непрерывно продолжалась, следствием чего явилась еще одна пересмотренная версия проекта, описанная в документе ptc/04-10-02. Наконец, на официальном сайте OMG (**www.omg.org**) в начале 2006 г. появились окончательно принятые спецификации на язык UML 2.0, который с этого момента приобретает статус "текущей официальной версии".

История разработки и последующего развития языка UML схематично представлена на рис. 1.7.

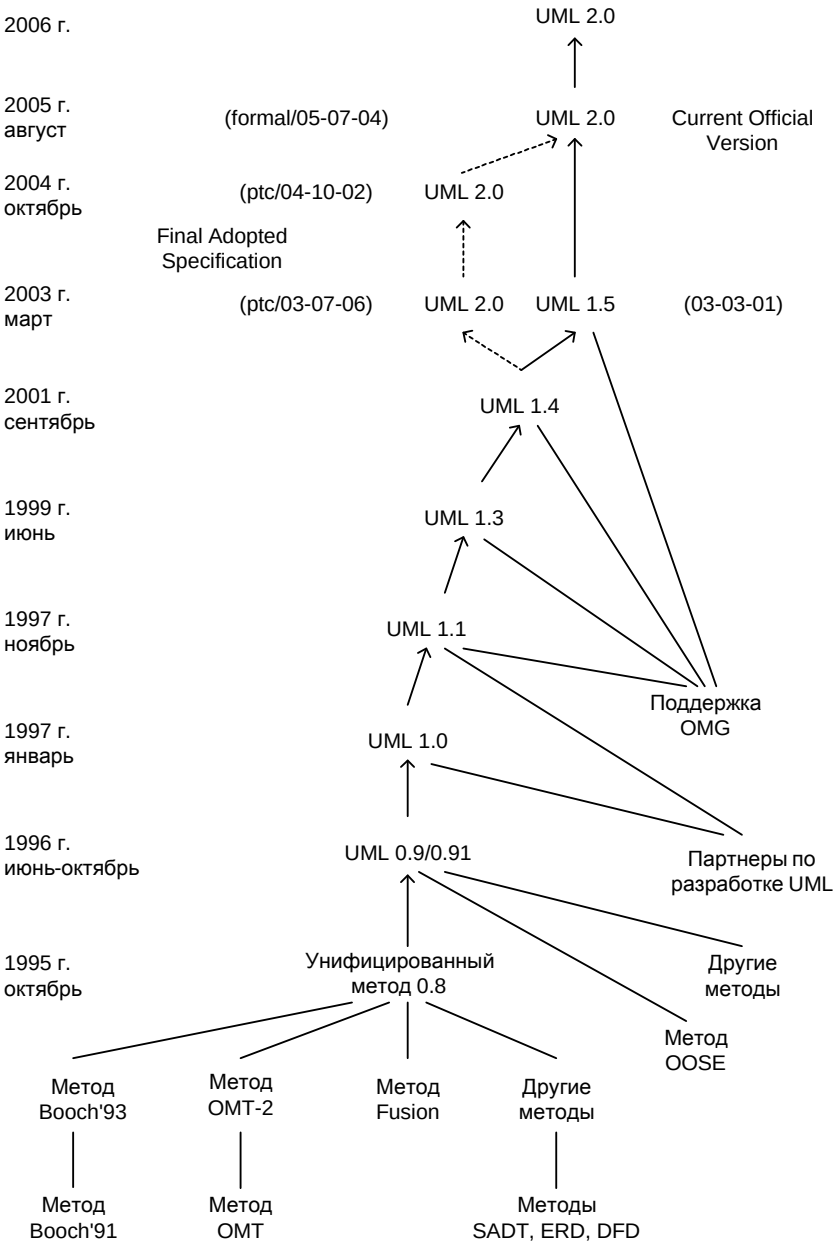


Рис. 1.7. История развития языка UML

Спецификация языка UML 2.0 описана в двух основных документах: formal/05-07-04 "Unified Modeling Language: Superstructure" и formal/05-07-05 "Unified Modeling Language: Infrastructure". Сам язык UML 2.0 не является чьей-либо собственностью и не запатентован кем-либо, хотя указанные выше документы защищены законом об авторском праве. В то же время аббревиатура UML 2.0, как и некоторые другие (OMG, CORBA, ORB), является торговой маркой их законных владельцев, о чем следует упомянуть в данном контексте.

Язык UML 2.0 ориентирован для применения в качестве языка моделирования различными пользователями и научными сообществами для решения широкого класса задач ООАП. Многие специалисты по методологии, организации и поставщики инструментальных средств используют эту версию языка UML в своих разработках еще в статусе "проекта". При этом термин "унифицированный" в названии UML 2.0 не является случайным и имеет два аспекта. С одной стороны, он фактически устраняет многие из несущественных различий между известными ранее языками моделирования и методиками построения диаграмм. С другой стороны, создает предпосылки для унификации различных моделей и этапов их разработки для широкого класса систем, не только программного обеспечения, но и бизнес-процессов. Семантика языка UML 2.0 определена таким образом, что она не является препятствием для последующих усовершенствований при появлении новых концепций моделирования.

В настоящее время на рынке CASE-средств представлено более десятка программных инструментов, поддерживающих нотацию языка UML 2.0 и обеспечивающих интеграцию, включая прямую и обратную генерацию кода программ, с наиболее распространенными языками и средами программирования, такими как MS Visual C++, C#, Java, MS Visual Basic, Object Pascal/Delphi.

Компания Borland, хотя и с некоторым опозданием, но весьма удачно подключилась к поддержке в своих продуктах нотации языка UML. В начале заметным событием стало появление в популярном RAD-средстве Borland Delphi 7 инструмента ModelMaker, предназначенного для работы с диаграммами языка UML 1.4 с последующей генерацией кода на Delphi Pascal. Эта программная разработка нидерландской компании ModelMaker Tools явилась одной из первых, в которой была реализована не только поддержка графической нотации языка UML с генерацией исполнимого кода, но и использование так называемых *паттернов* проектирования, которые еще несколько лет назад вызывали в лучшем случае улыбку, а в худшем — откровенный скепсис у программистов.

После приобретения компании Together Soft поддержка языка UML становится самостоятельным направлением развития инструментов моделирования

для компании Borland. В этом контексте наиболее значительным событием стало появление в 2005 г. целой линейки продуктов Borland® Together 2005, среди которых оказался и Borland Together Designer 2005, поддерживающим нотацию языка UML 2.0 в статусе проекта. Это событие сразу же было отмечено всем сообществом пользователей языка UML, поскольку аналогичные средства от IBM Rational появились почти на год позже.

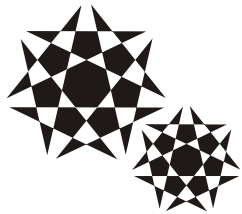
На момент написания книги Together Designer 2005 интегрирован в Borland Developer Studio 2006, предоставляя совершенно уникальные возможности двунаправленной разработки программных приложений с инструментальной поддержкой двух основных версий языка UML 1.5 и 2.0. Дополнительная интеграция с библиотекой компонентов ECO III предоставляет исключительную возможность разработки реальных приложений на основе визуальных моделей в нотации UML практически без написания программного кода. Наконец, в этом инструменте нашла практическое воплощение диаграмма конечного автомата, которая из инструмента виртуального моделирования превратилась в конкретный способ управления временем жизни программных объектов.

Подводя итог анализу методологии ООАП, истории появления и развития языка UML 2.0, можно сделать следующие выводы.

Во-первых, имеются все основания предполагать, что в ближайшие годы интерес к языку UML 2.0 со стороны самых различных специалистов будет все более и более возрастать. Язык UML 2.0 не только станет основой для разработки и реализации в новых версиях популярных RAD-средств, но и в CASE-средствах визуального и имитационного моделирования.

Во-вторых, заложенные в языке UML 2.0 потенциальные возможности могут быть использованы как для объектно-ориентированного анализа и проектирования программных систем, так и для моделирования и документирования бизнес-процессов, а в более широком контексте — для представления знаний в интегрированных интеллектуальных системах, которыми, по существу, станут перспективные сложные программно-технологические комплексы.

Глава 2



Основные элементы нотации языка UML 2.0

Язык UML 2.0 обладает весьма широкими возможностями, которые предназначены для решения разнообразных задач визуального моделирования из самых различных прикладных областей. Однако, обладая значительной избыточностью, не все его возможности моделирования являются обязательно необходимыми при разработке конкретных приложений. Соответствующая согласованность используемых конструкций языка UML 2.0 обеспечивается его модульным построением. Все это позволяет разработчикам выбирать только те элементы, которые представляют прямой интерес в контексте разрабатываемого проекта.

Язык UML 2.0 основан на некотором числе базовых понятий, которые могут быть изучены и применены большинством программистов и разработчиков, знакомых с методами объектно-ориентированного анализа и проектирования. При этом базовые понятия могут комбинироваться и расширяться таким образом, что специалисты объектного моделирования получают возможность самостоятельно разрабатывать модели больших и сложных систем в самых различных областях приложений.

2.1. Назначение языка UML 2.0

Язык UML 2.0, также как и его предыдущие версии, предназначен для решения следующих задач.

- Предоставить в распоряжение всех пользователей легко воспринимаемую и выразительную нотацию для визуального моделирования, специально предназначенную для разработки и документирования моделей сложных систем самого различного целевого назначения.
  - Речь идет о том, что важным фактором дальнейшего развития и повсеместного использования методологии ООАП является интуитивная

ясность и понятность основных конструкций соответствующего языка моделирования. Язык UML 2.0 включает в себя не только абстрактные конструкции, для представления метамоделей систем, но и целый ряд конкретных понятий, имеющих вполне определенную семантику. Это позволяет языку UML 2.0 одновременно достичь не только универсальности представления моделей для самых различных приложений, но и возможности описания достаточно тонких деталей реализации этих моделей применительно к конкретным системам.

- Для адекватного понимания базовых конструкций языка UML 2.0 важно не только владеть навыками объектно-ориентированного программирования, но и хорошо представлять себе общую проблематику процесса разработки моделей систем. Именно интеграция этих представлений образует новую парадигму в методологии ООАП, практическим следствием и центральным стержнем которой является язык UML 2.0.

❒ Снабдить исходные понятия языка UML 2.0 возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области.

- Хотя язык UML 2.0 является формальным языком спецификаций, формальность его описания отличается от синтаксиса как традиционных формально-логических языков, так и известных языков программирования. Разработчики из OMG предполагают, что язык UML 2.0 как никакой другой может быть приспособлен для конкретных предметных областей. Это становится возможным по той причине, что в самом описании языка UML 2.0 заложен механизм расширения базовых понятий, который является самостоятельным элементом и имеет собственное описание в форме правил расширения.
- В то же время разработчики из OMG считают крайне нежелательным переопределение базовых понятий языка. Это может привести к неоднозначной интерпретации их семантики и возможной путанице. Базовые понятия языка UML 2.0 не следует изменять больше, чем это необходимо для их расширения. Все пользователи должны быть способны строить модели систем для большинства обычных приложений с использованием только базовых конструкций языка UML 2.0 без применения механизма расширения. При этом новые понятия и графические обозначения целесообразно применять только в тех ситуациях, когда имеющихся базовых конструкций явно недостаточно для построения моделей системы.
- Язык UML 2.0 допускает также специализацию базовых понятий. Речь идет о том, что в конкретных приложениях пользователи должны уметь

дополнять имеющиеся базовые понятия новыми характеристиками или свойствами, которые не противоречат семантике этих понятий.

- Описание языка UML 2.0 должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.
  - Речь идет о том, что конструкции языка UML 2.0 не должны зависеть от особенностей их реализации на той или иной платформе или языке программирования. Другими словами, хотя отдельные понятия языка UML 2.0 семантически связаны с последними, их жесткая интерпретация в форме конструкций программирования не может быть признана корректной. Разработчики из OMG считают необходимым свойством языка UML 2.0 его контекстно-программную и платформенную независимость.
  - С другой стороны, язык UML 2.0 должен обладать потенциальной возможностью реализации своих конструкций на том или ином языке программирования. Конечно, в первую очередь имеются в виду языки, поддерживающие концепцию ООП, такие как Java, C++, C#, Object Pascal/Delphi. Именно это свойство языка UML 2.0 делает его современным и конструктивным средством решения задач моделирования сложных систем. В то же время, предполагается, что для программной поддержки конструкций языка UML 2.0 могут быть разработаны специальные инструментальные CASE-средства. Наличие последних имеет принципиальное значение для широкого распространения и использования языка UML 2.0.
- Описание языка UML 2.0 должно включать в себя семантический базис для понимания общих особенностей ООАП.
  - Говоря об этой особенности, имеют в виду самодостаточность языка UML 2.0 для понимания не только его базовых конструкций, но что не менее важно — понимания общих принципов ООАП. В этой связи необходимо отметить, что поскольку язык UML 2.0 не является языком программирования, а служит средством для решения задач объектно-ориентированного моделирования систем, описание языка UML 2.0 должно по возможности включать в себя все необходимые понятия методологии ООАП.
  - С другой стороны, какие бы то ни было ссылки на дополнительные источники, содержащие важную для понимания языка UML 2.0 информацию, по мнению разработчиков из OMG, должны быть исключены. Это позволит избежать неоднозначного толкования принципиальных особенностей методологии ООАП и их реализации в форме базовых конструкций языка UML 2.0.

- ❒ Поощрять развитие рынка программных инструментальных средств, поддерживающих методологию ООАП и концепцию MDA.
  - Более 800 ведущих производителей программных и аппаратных средств, усилия которых сосредоточены в рамках OMG, видят перспективы развития современных информационных технологий и основу своего коммерческого успеха в широком продвижении на рынок инструментальных средств, поддерживающих объектные технологии. С этой точки зрения языку UML 2.0 отводится роль базового концептуального средства для описания и документирования различных объектных технологий.
- ❒ Способствовать распространению объектных технологий и соответствующих понятий ООАП.
  - Если попытаться составить объективную картину возможностей языка UML 2.0, то можно прийти к следующему заключению. Следует признать, что усилия достаточно большой группы разработчиков, направленные на интеграцию в рамках языка UML 2.0 многих известных техник визуального моделирования, привели к серьезному усложнению языка UML 2.0 по сравнению с другими известными нотациями и его предыдущими версиями. Платой за сложность является действительно высокая гибкость и широкие изобразительные возможности языка UML 2.0. В свою очередь, использование языка UML 2.0 для решения всевозможных практических задач будет только способствовать его дальнейшему совершенствованию, а значит, и дальнейшему развитию объектных технологий и практики ООАП.
- ❒ Интегрировать в себя новейшие и наилучшие достижения практики ООАП.

  Язык UML непрерывно совершенствуется разработчиками, и основой этой работы является его дальнейшая интеграция с современными технологиями моделирования. При этом различные методы системного моделирования получают свое прикладное осмысление в контексте методологии ООАП. В последующем эти методы могут быть включены в состав очередных версий языка UML в форме дополнительных базовых понятий, наиболее адекватно и полно отражающие наилучшие достижения практики ООАП.

Чтобы решить указанные выше задачи, за свою историю язык UML претерпел определенную эволюцию. В результате описание самого языка UML 2.0 стало нетривиальным, поскольку семантика базовых понятий включает в себя целый ряд перекрестных связей с другими понятиями и конструкциями языка. В связи с этим последовательное рассмотрение основных конструкций

языка UML 2.0 стало практически невозможным, т. к. одни и те же понятия могут использоваться при построении различных диаграмм и представлений. В то же время каждое из представлений модели обладает собственными семантическими особенностями, которые накладывают отпечаток на семантику базовых понятий языка в целом.

Как результат, разработчикам нет необходимости знать весь язык UML 2.0, чтобы эффективно его использовать. Кроме того, большинство единиц языка может быть разделено на многократные приращения, каждое из которых добавляет к предыдущим дополнительные возможности моделирования. Такая дискретная декомпозиция языка UML 2.0 предназначена для того, чтобы сделать более легким его изучение и использование на практике. При этом отдельные приращения описываются в форме слияния пакетов, что находит свое выражение при рассмотрении отдельных типов канонических диаграмм.

Учитывая эти особенности, принятая в книге последовательность изучения языка UML 2.0 основывается на рассмотрении основных графических средств моделирования, а именно — канонических диаграмм. Все понятия, необходимые для построения соответствующих диаграмм, вводятся по мере необходимости. В данной главе приводится описание общих особенностей языка UML 2.0, которые в той или иной степени оказывают влияние на понимание всех его базовых конструкций и канонических диаграмм.

2.2. Общая структура языка UML 2.0

С самой общей точки зрения описание языка UML 2.0 состоит из двух взаимодействующих частей: семантики и нотации.

Семантика (semantics) — система правил и соглашений, определяющих толкование и придание смысла конструкциям некоторого языка.

Нотация (notation) — система условных обозначений, принятая в некотором языке для изображения и визуализации модели.

В рамках языка UML 2.0 семантика и нотация тесно взаимосвязаны. Семантика языка UML 2.0 описывается с использованием некоторого подмножества нотации UML 2.0. В свою очередь, нотация UML 2.0 описывает соответствие или отображение графической нотации в базовые понятия семантики. Таким образом, с функциональной точки зрения эти две части дополняют друг друга. При этом семантика языка UML 2.0 описывается на основе некоторой метамодели, имеющей три отдельных представления: абстрактный

синтаксис, правила корректного построения выражений и семантику. Рассмотрение семантики языка UML 2.0 предполагает некоторый "полуформальный" стиль изложения, который объединяет естественный и формальный языки для представления базовых понятий и правил их расширения.

Семантика определяется для двух категорий объектных моделей: структурных моделей (статических моделей) и моделей поведения (динамических моделей).

Структурные модели (structured models) — модели, предназначенные для описания статической структуры сущностей или элементов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения.

Модели поведения (behavioral models) — модели, предназначенные для описания процесса функционирования элементов системы, включая их методы и взаимодействие между ними, а также процесс изменения состояний отдельных элементов и системы в целом.

Для решения самого широкого диапазона задач моделирования в языке UML 2.0 разработана достаточно полная семантика для всех элементов графической нотации. При этом требования семантики языка UML 2.0 конкретизируются при построении отдельных типов диаграмм, последовательное рассмотрение которых служит темой *части II* книги. Нотация языка UML 2.0 включает в себя описание отдельных семантических элементов, которые могут применяться при построении диаграмм.

Формальное описание самого языка UML 2.0 основывается на некоторой общей иерархической структуре модельных представлений, состоящей из четырех уровней:

- мета-метамодель (M3);
- метамодель (M2);
- модель (M1);
- объекты или экземпляры (M0).

Уровень *мета-метамодели* образует исходную формально-логическую основу для всех возможных метамодельных представлений. Главное предназначение этого уровня состоит в том, чтобы определить язык для спецификации метамодели. Мета-метамодель определяет модель языка UML 2.0 на самом высоком уровне абстракции и является наиболее компактным ее описанием. С другой стороны, мета-метамодель может специфицировать несколько метамоделей, чем достигается потенциальная гибкость включения дополни-

тельных понятий. Хотя в книге этот уровень не рассматривается, он наиболее тесно связан с теорией формальных языков. Примерами понятий этого уровня служат метакласс, метаатрибут, метаоперация.

Метамодель является экземпляром или конкретизацией мета-метамодели. Главная задача этого уровня — определить язык для спецификации моделей. Данный уровень является более конструктивным, чем предыдущий, поскольку обладает более развитой семантикой базовых понятий. Все основные понятия языка UML 2.0 — это понятия уровня метамодели. Примерами таких понятий являются класс, атрибут, операция, компонент, ассоциация и многие другие. Рассмотрению семантики и графической нотации понятий уровня метамодели собственно и посвящена настоящая книга.

Модель в контексте языка UML 2.0 является экземпляром метамодели в том смысле, что любая конкретная модель системы должна использовать только понятия метамодели, конкретизировав их применительно к данной ситуации. Это уровень для описания информации о конкретной предметной области. Однако поскольку при построении модели используются понятия языка UML 2.0, то необходима полная согласованность понятий уровня модели с базовыми понятиями языка UML 2.0 уровня метамодели. Примерами понятий уровня модели могут служить, например, имена полей проектируемой базы данных, такие как: имя и фамилия сотрудника, возраст, должность, адрес, телефон. При этом данные понятия используются лишь как имена соответствующих информационных атрибутов.

Конкретизация понятий модели происходит на уровне *объектов* или *экземпляров*. При этом совокупность объектов рассматривается как отдельный экземпляр модели времени выполнения, поскольку содержит конкретную информацию относительно того, чему в действительности соответствуют те или иные понятия модели. Примером объекта может служить следующая запись в проектируемой базе данных: "Игорь Греков, 40 лет, модельер, ул. Системная, 1-2, 123-4567". Рассмотренную 4-уровневую структуру модельных представлений языка UML 2.0 можно проиллюстрировать следующим образом (рис. 2.1).

Описание семантики языка UML 2.0 предполагает рассмотрение базовых понятий только уровня *метамодели (M2)*, который представляет собой пример или частный случай уровня мета-метамодели. Метамодель UML 2.0 является по своей сути скорее логической моделью, чем физической или моделью реализации. Особенность логической модели заключается в том, что она концентрирует внимание на декларативной или концептуальной семантике, опуская детали конкретной физической реализации моделей. При этом отдельные реализации, использующие данную логическую метамодель, должны быть согласованы с ее семантикой, а также поддерживать возможности импорта и экспорта отдельных логических моделей.

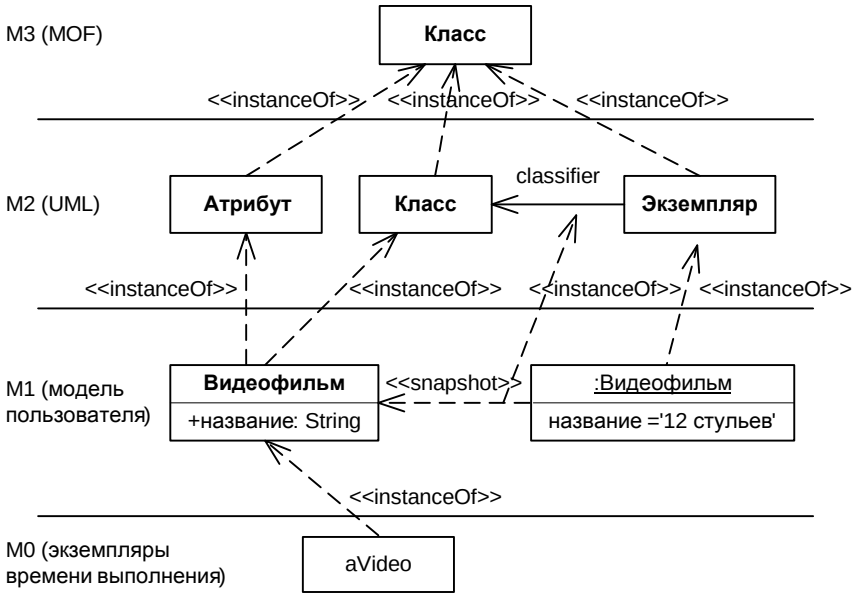


Рис. 2.1. Иллюстрация 4-уровневой структуры модельных представлений языка UML 2.0

В то же время, логическая метамодель может быть реализована различными способами для обеспечения требуемого уровня производительности и надежности соответствующих инструментальных средств. В этом заключается недостаток логической модели, которая не содержит на уровне семантики требований, обязательных для ее эффективной последующей реализации. Однако согласованность метамодели с конкретными моделями реализации является обязательной для всех разработчиков программных средств, обеспечивающих поддержку языка UML 2.0.

Метамодель языка UML 2.0 имеет довольно сложную структуру, которая включает в себя метаклассы, метаассоциации и стереотипы, число которых возрастает с появлением новых версий языка. Чтобы справиться с этой сложностью языка UML 2.0, все его элементы организованы специальным образом в логически связанные подмножества, получившие название пакетов. Поэтому рассмотрение языка UML 2.0 на метамодельном уровне заключается в описании его наиболее общих логических блоков или *пакетов*.

Говоря о пакетах в контексте общего описания языка, мы, по сути дела, приступаем к рассмотрению графической нотации языка UML 2.0. При этом следует заметить, что для описания языка UML 2.0 используются средства самого этого языка. Одним из таких средств является *пакет*, который в общем случае служит для группировки любых элементов модели. Пакеты могут находиться в различных отношениях и быть вложенными друг в друга.

2.3. Пакеты в языке UML 2.0

Пакет — основной способ организации элементов метамодели языка UML 2.0. Каждый пакет владеет всеми своими элементами, т. е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. В языке UML 2.0 для визуализации пакетов разработана специальная символика или графическая нотация, которой мы и будем пользоваться в дальнейшем. Именно с описания этой системы обозначений мы приступим к изучению основных элементов языка.

Для графического изображения пакетов на диаграммах применяется специальный графический символ — большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны первого (рис. 2.2). Визуально символ пакета напоминает пиктограмму папки на экране монитора. Внутри большого прямоугольника могут изображаться элементы, которые принадлежат этому пакету. Если такие элементы не указаны, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели (рис. 2.2, *а*). Если же элементы пакета указаны, то имя пакета записывается в верхнем маленьком прямоугольнике (рис. 2.2, *б*).

Рис. 2.2. Графическое изображение пакетов в языке UML 2.0

Говоря об имени пакета, следует остановиться на общем соглашении об именах в языке UML 2.0. В данном случае именем пакета может быть строка текста, содержащая произвольное число букв, цифр и некоторых специальных знаков. С целью удобства спецификации пакетов принято в качестве их имен использовать существительные, возможно, с пояснительными словами, например, контроллер, графический интерфейс, форма ввода данных.

Сами по себе пакеты находят ограниченное применение, поскольку содержат лишь информацию о входящих в их состав элементах модели. Не менее важно представить графически отношения, которые могут иметь место между отдельными пакетами. Как и в теории графов, для визуализации отношений

в языке UML 2.0 применяются отрезки линий, внешний вид которых имеет смысловое содержание или семантику.

Одним из типов отношений между пакетами является отношение вложенности или включения пакетов друг в друга. С одной стороны, в языке UML 2.0 это отношение может быть изображено без использования линий простым размещением одного пакета-прямоугольника внутри другого пакета-прямоугольника (рис. 2.3). Так, в данном случае `Пакет A` содержит в себе два подпакета: `Пакет B` и `Пакет C`.

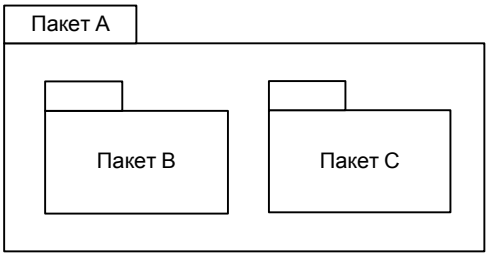


Рис. 2.3. Графическое изображение вложенности пакетов

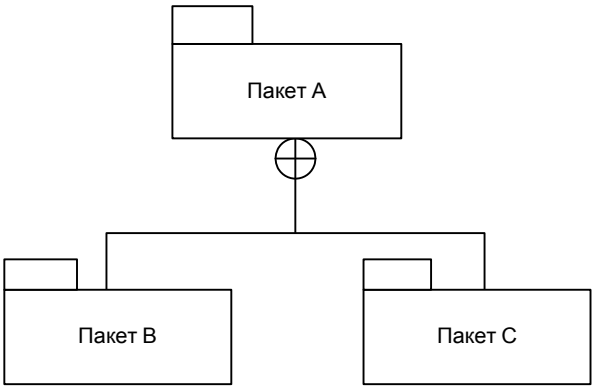


Рис. 2.4. Альтернативное графическое изображение вложенности пакетов

С другой стороны, это же отношение может быть изображено с помощью отрезков линий аналогично графическому представлению дерева. В этом случае пакет, содержащий другие пакеты, или пакет-контейнер изображается в верхней части рисунка, а его подпакеты — уровнем ниже. Пакет-контейнер соединяется с подпакетами сплошной линией, на конце которой, примыкающей к пакету контейнеру, изображается специальный символ — ⊕. Этот сим-

вол означает, что подпакеты являются "собственностью" или частью контейнера, и, кроме этих подпакетов, контейнер не содержит никаких других подпакетов. Рассмотренный выше пример (см. рис. 2.3) может быть представлен альтернативно следующим образом (рис. 2.4).

На графических диаграммах между пакетами могут указываться и другие типы отношений, которые будут описаны при рассмотрении диаграммы пакетов *(см. главу 6)*.

2.4. Основные пакеты метамодели языка UML 2.0

Основой представления языка UML 2.0 на метамодельном уровне является спецификация его основных логических частей в форме пакетов: `Элементы ядра`, `Абстракции`, `Основы`, `Конструкции` и `Простейшие Типы` (рис. 2.5).

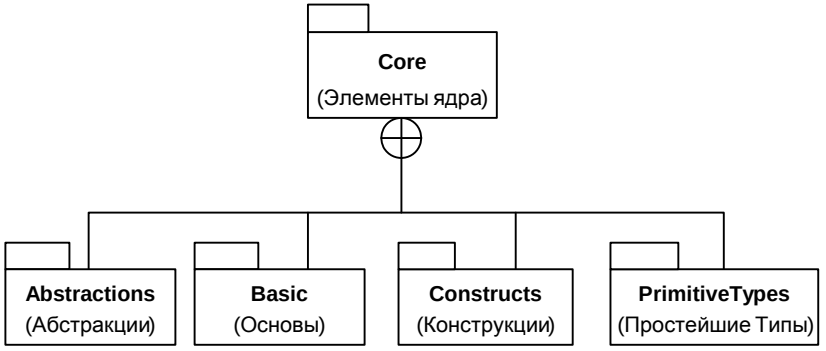


Рис. 2.5. Основные пакеты метамодели языка UML 2.0

Эти пакеты, в свою очередь, также включают в себя другие подпакеты, которые содержат описание фундаментальных понятий языка UML 2.0. Далее дается краткая характеристика элементов каждого из перечисленных подпакетов, входящих в состав пакета `Элементы ядра`. Более полное рассмотрение отдельных пакетов метамодели будет представлено в главах, посвященных изучению отдельных видов канонических диаграмм. Последние аккумулируют в себе не только различные представления моделируемой системы, но и более детально раскрывают семантические особенности применения базовых конструкций языка UML 2.0 в процессе построения конкретных моделей.

2.4.1. Пакет *Абстракции*

Пакет Абстракции является наиболее фундаментальным из всех подпакетов, которые входят в пакет Элементы ядра языка UML 2.0. Он определяет основные абстрактные элементы, необходимые для разработки объектных моделей. При этом абстрактные элементы метамодели не имеют экземпляров или примеров и используются исключительно для уточнения других элементов модели.

В этот пакет входят основные подпакеты языка UML 2.0: Elements (Элементы), Ownerships (Права собственности), Namespaces (Пространства имен), Comments (Комментарии), Relationships (Отношения), Visibilities (Видимости), Expressions (Выражения), StructuralFeatures (Структурные характеристики), BehavioralFeatures (Характеристики поведения), Classifiers (Классификаторы), Constraints (Ограничения), Generalizations (Обобщения), Instances (Экземпляры), MultiplicityExpressions (Выражения кратности), Redefinitions (Переопределения), Changeabilities (Способности к изменениям), Super (Верхний уровень), Literals (Литералы), Multiplicities (Кратности), TypedElements (Типизированные элементы).

2.4.2. Пакет *Основы*

Пакет Основы содержит минимальное подмножество элементов языка UML 2.0, которые называются *метаклассами* и используются для дальнейшей конкретизации при моделировании структуры систем. Метаклассы данного пакета и отношения между ними описывают следующие диаграммы:

- диаграмма типов, которая специфицирует метаклассы, имеющие отношение к именованным и типизированным элементам: Type (Тип), Element (Элемент), TypedElement (Типизированный элемент), NamedElement (Именованный элемент), Comment (Комментарий);
- диаграмма классов, которая определяет конструкции, необходимые для моделирования классов: Class (Класс), Property (Свойство), Operation (Операция), Parameter (Параметр), MultiplicityElement (Элемент кратности);
- диаграмма типов данных, которая специфицирует метаклассы, которые определяют типы данных: Type (Тип), PrimitiveType (Простейший Тип), Enumeration (Перечисление), DataType (Тип Данных);
- диаграмма пакетов, которая определяет базовые конструкции, имеющие отношение к пакетам и их содержимому: Package (Пакет) и Type (Тип).

2.4.3. Пакет *Конструкции*

Пакет Конструкции определяет расширенное подмножество элементов языка UML 2.0, которые используются для детального моделирования систем. Для описания метаклассов данного пакета используются следующие диаграммы:

- диаграмма верхнего уровня (Root diagram), которая специфицирует следующие конструкции: Element (Элемент), Relationship (Отношение), DirectedRelationship (Направленное отношение) и Comment (Комментарий);
- диаграмма выражений (Expressions diagram), которая специфицирует следующие конструкции: Expression (Выражение), ValueSpecification (Спецификация значения) и OpaqueExpression (Неопределенное выражение);
- диаграмма классов (Classes diagram), которая специфицирует следующие конструкции: Association (Ассоциация), Class (Класс) и Property (Свойство), а также добавляет характеристики к конструкциям: Classifier (Классификатор) и Operation (Операция);
- диаграмма классификаторов (Classifiers diagram), которая специфицирует следующие конструкции: Classifier (Классификатор), TypedElement (Типизированный элемент), MultiplicityElement (Элемент кратности), RedefinableElement (Переопределяемый элемент), Feature (Характеристика) и StructuralFeature (Структурная характеристика). Каждое из этих понятий расширяется и переопределяется по сравнению с их определением в пакетах Basic и Abstractions;
- диаграмма ограничений (Constraints diagram), которая специфицирует конструкцию Constraint (Ограничение) и добавляет характеристики к конструкции Namespace (Пространство имен);
- диаграмма типов данных (DataTypes diagram), которая специфицирует следующие конструкции: DataType (Тип данных), Enumeration (Перечисление), EnumerationLiteral (Литерал перечисления) и PrimitiveType (Простейший Тип), а также добавляет характеристики к конструкциям: Property (Свойство) и Operation (Операция). Эти конструкции используются для определения простейших типов данных (таких как Integer и String), а также определенных пользователем перечислимых типов данных;
- диаграмма пространств имен (Namespaces diagram), которая специфицирует конструкцию Namespace (Пространство имен). Эта диаграмма определяет именованные элементы как элементы пространства имен, а также специфицирует общую способность любого пространства имен импортировать все или отдельные элементы пакетов;

- диаграмма операций (Operations diagram), которая специфицирует следующие конструкции: `BehavioralFeature` (`Характеристика поведения`), `Operation` (`Операция`) и `Parameter` (`Параметр`);
- диаграмма пакетов (Packages diagram), которая специфицирует следующие конструкции: `Package` (`Пакет`) и `PackageMerge` (`Слияние пакетов`).

2.4.4. Пакет *Простейшие Типы*

Пакет `Простейшие Типы` содержит несколько предопределенных типов, которые используются при определении абстрактного синтаксиса метамоделей. Они включают в себя такие простейшие типы, как `Integer`, `Boolean`, `String` и `UnlimitedNatural`. Эти типы могут также включаться в модели пользователей (рис. 2.6).

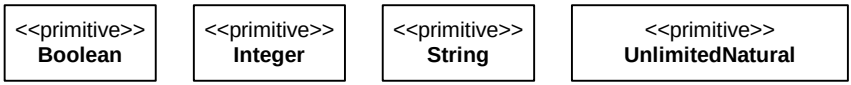


Рис. 2.6. Содержимое пакета `Простейшие Типы`

В метамодели UML 2.0 простейшие типы используются для объявления типов атрибутов классов. Они могут записываться в форме соответствующих строк текста на диаграммах и не иметь отдельного значка "тип данных". Благодаря этому происходит уменьшение размеров диаграмм без потери информации. Однако каждая из одинаковых записей для некоторого типа данных должна соответствовать одному и тому же типу данных в модели. При этом типы данных, используемые в описании языка UML 2.0, могут отличаться от типов данных, которые определяет разработчик для своей модели на языке UML 2.0. Типы данных в последнем случае будут являться частным случаем или экземплярами метакласса `Типы Данных`, который также определен в метамодели.

Boolean (Логический)

Логический тип используется для спецификации логического выражения и содержит два предварительно определенных значения: "истина" и "ложь". Логический тип является экземпляром `Простейшего Типа`. В метамодели `Логический тип` определяет перечисление, литералами которого являются значения:

- `true` (истина), которое соответствует выполнению некоторого логического условия;
- `false` (ложь), которое соответствует невыполнению некоторого логического условия.

Логический тип может использоваться в качестве типа атрибутов в метамодели. Экземпляры логического типа могут быть значениями, ассоциированными со слотами, которые могут иметь только одно из значений литералов: "истина" или "ложь". Этот тип используется также для логических атрибутов и логических выражений, например, в выражениях OCL. Пример использования `Логического типа` в качестве типа значений атрибута класса изображен на рис. 2.7.

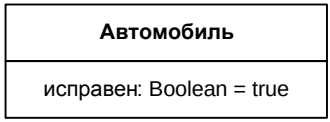


Рис. 2.7. Пример логического атрибута класса

Integer (Целочисленный)

Целочисленный тип является простейшим типом для представления целочисленных значений. Экземпляр `Целочисленного типа` является элементом из бесконечного множества целых чисел (..., –2, –1, 0, 1, 2, ...). Целочисленный тип может представляться в форме типа атрибутов в метамодели. Экземпляры `Целочисленного типа` должны быть значениями, например, такими как: 1, –5, 2, 34, 26 524 и т. д. Пример использования `Целочисленного типа` в качестве типа значений атрибута класса изображен на рис. 2.8.

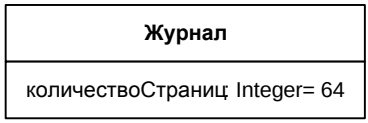


Рис. 2.8. Пример целочисленного атрибута класса

String (Строка)

Тип `Строка` является последовательностью символов из некоторого подходящего множества знаков, используемого для отображения информации о модели. Множества символов могут включать в себя произвольные алфавиты и символы. Экземпляр типа `Строка` определяет некоторую часть текста. Семантика самой строки зависит от ее назначения — она может быть комментарием, выражением языка вычислений, выражением OCL и т. д. Она ис-

пользуется для строковых атрибутов и строковых выражений в метамодели. Строка может использоваться в качестве типа атрибутов в самой метамодели. Значение строки является последовательностью символов, заключенных в двойные кавычки (").

Предполагается, что базовое множество символов является достаточным для представления произвольных символов в различных языках. В частности, традиционное 8-битовое множество символов ASCII является недостаточным для этой цели. Предполагается также, что средства и компьютеры манипулируют и хранят строки правильно, а также могут использовать строки произвольной длины.

Строка изображается графически в форме строки текста. Обычные печатные символы должны изображаться непосредственно. Изображение символов, которые не могут быть напечатаны, в языке UML 2.0 не специфицировано и является платформенно-зависимым. Пример использования типа `Строка` в качестве типа значений атрибута класса изображен на рис. 2.9.

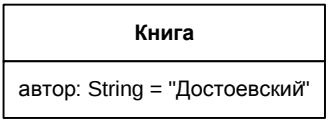


Рис. 2.9. Пример строкового атрибута класса

UnlimitedNatural (Неограниченное натуральное число)

Тип `Неограниченное натуральное число` является простейшим типом для представления значения произвольного натурального числа. Экземпляр `Неограниченного натурального числа` является элементом из бесконечного множества натуральных чисел (0, 1, 2, ...). Тип `Неограниченное натуральное число` используется для указания верхних границ кратности элементов в метамодели. Экземпляры `Неограниченного натурального числа` должны быть значениями, такими как 1, 5, 398 475 и т. д. Значение бесконечности может быть изображено с использованием звездочки (`*`). Пример использования типа `Неограниченное натуральное число` для указания верхней границы кратности конца ассоциации изображен на рис. 2.10.

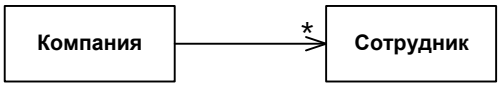


Рис. 2.10. Пример использования неограниченного натурального числа в качестве границы кратности

При задании пользовательских типов данных наиболее часто применяется неформальная конструкция, которая получила название *перечисления*. Речь идет о множестве допустимых значений атрибута, которое наделяется некоторым отношением порядка. При этом упорядоченность значений либо указывается явно заданием первого и последнего элементов списка, либо следует неявно в случае простейшего типа данных, как, например, для множества натуральных чисел. В пакете `Типы данных` определены способы спецификации перечислений для корректного задания допустимых значений атрибутов.

2.4.5. Пакет *Модели*

Пакет `Модели` (`Models`) входит в состав вспомогательных конструкций и специфицирует базовые элементы языка UML 2.0, которые необходимы для построения отдельных модельных представлений. В нем определяется семантика понятия модели (Model) и ее графическая нотация в форме пакета. Поскольку пакет является метаклассом в языке UML 2.0, он может быть использован для группировки других элементов модели, таких как классификаторы, ассоциации и другие пакеты.

`Модель` в нотации языка UML 2.0 является специальным видом пакета и представляет абстракцию физической системы, которая предназначена для вполне определенной цели. Именно эта цель предопределяет те элементы, которые должны быть включены в модель, и те, рассмотрение которых не является обязательным. Другими словами, модель отражает релевантные аспекты физической системы, оказывающие непосредственное влияние на достижение поставленной цели. В прикладных задачах цель обычно задается в форме исходных требований к системе, которые, в свою очередь, в языке UML 2.0 могут быть записаны в форме отдельных вариантов использования.

В языке UML 2.0 для одной и той же физической системы могут быть определены разные модели, каждая из которых специфицирует систему с различных точек зрения. Примерами таких моделей являются логическая модель, модель проектирования, модель анализа, модель вариантов использования и др. При этом каждая такая модель имеет свою собственную точку зрения на физическую систему и свой собственный уровень абстракции. Модели,

как и пакеты, могут быть вложенными друг в друга. Со своей стороны, пакет может включать в себя несколько различных моделей одной и той же системы, и в этом состоит один из важнейших механизмов разработки моделей на языке UML 2.0.

Модель в нотации UML 2.0 может быть представлена с использованием символа пакета с небольшим треугольником внутри прямоугольника. Если внутри пакета изображается содержимое модели, треугольник может быть изображен справа от имени модели во вкладке. Так, например, если `Модель системы` в контексте языка UML 2.0 включает в себя `Модель анализа` и `Модель проектирования`, то это может быть изображено следующим образом (рис. 2.11).

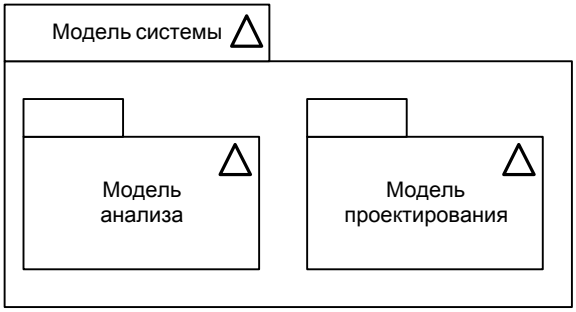


Рис. 2.11. Изображение `Модели системы` в форме двух пакетов или подмоделей

Альтернативно модель может быть также изображена в форме символа обычного пакета с дополнительным ключевым словом `<<model>>`, расположенном выше имени соответствующей модели.

2.5. Особенности спецификации метамодели языка UML 2.0

Метамодель языка UML 2.0 описывается на некотором полуформальном языке с использованием трех видов представлений:

- ❒ абстрактный синтаксис;
- ❒ правила правильного построения выражений;
- ❒ семантика.

Абстрактный синтаксис представляет собой модель для описания некоторой части языка UML 2.0, предназначенной для построения диаграмм классов на основе описаний систем на естественном языке. Возможности абст-

рактного синтаксиса в языке UML 2.0 довольно ограничены и имеют отношение только к интерпретации обозначений отдельных элементов диаграмм и связей между элементами. К элементам абстрактного синтаксиса относятся некоторые ключевые слова и значения отдельных атрибутов базовых понятий уровня метамодели, которые имеют фиксированное обозначение в виде текста на естественном языке.

Правила правильного построения выражений используются для задания дополнительных ограничений или свойств, которыми должны обладать те или иные элементы модели. Для задания таких инвариантных свойств элементов и отношений между ними необходимо использовать специальные выражения некоторого формального языка, который в рамках UML 2.0 получил название *языка объектных ограничений* (Object Constraint Language, OCL).

Семантика языка UML 2.0 описывается в основном на естественном языке, но может включать в себя некоторые дополнительные обозначения, вытекающие из связей определяемых понятий с другими понятиями. Семантика понятий раскрывает их смысл или содержание. Сложность описания семантики языка UML 2.0 заключается именно в метамодельном уровне представлений его основных конструкций. С одной стороны, понятия языка UML 2.0 имеют абстрактный характер (например, ассоциация, композиция, агрегация, состояние, деятельность). С другой стороны, каждое из этих понятий допускает свою конкретизацию на уровне модели (сотрудник, отдел, должность, стаж).

Сложность описания семантики языка UML 2.0 вытекает из этой двойственности понятий, а понимание семантики носит индуктивный характер и требует для своей интерпретации примеров уровня модели и объекта. Иллюстрация абстрактных понятий на примере конкретных свойств и отношений, а также их значений позволяет акцентировать внимание на общих инвариантах этих понятий, что необходимо для понимания их семантики.

Метамодель языка UML 2.0 может рассматриваться как комбинация графической нотации (специальных обозначений), некоторого формального языка и естественного языка. При этом следует иметь в виду, что существует некоторый теоретический предел, который ограничивает описание метамодели средствами самой метамодели. Именно в подобных случаях используется естественный язык, обладающий наибольшими выразительными возможностями.

Хотя сам термин "естественный язык" далеко не однозначен и порождает целый ряд дополнительных вопросов, здесь мы ограничимся его трактовкой в форме обычного текста на русском и, возможно, английском языках.

Следует отметить, что для отечественных разработчиков полностью избежать использования английского языка при описании и использовании нотации UML 2.0 не удастся. Тем не менее, если исключить написание стандартных элементов и некоторых ключевых слов, то во всех остальных случаях под естественным языком можно понимать русский язык без специальных оговорок.

Для придания формального характера моделям UML 2.0 использование естественного языка должно строго соответствовать определенным правилам. Например, описание семантики языка UML 2.0 может включать в себя фразы типа "Сущность *А* обладает способностью..." или "Сущность *Б* есть сущность *В*". В каждом из этих случаев мы будем понимать смысл фраз, руководствуясь традиционным пониманием предложений русского языка. Однако этого может оказаться недостаточным для более формального представления знаний о рассматриваемых сущностях. Тогда необходимо дополнительно специфицировать семантику этих простых фраз, для чего целесообразно использовать следующие правила:

- явно указывать в тексте экземпляр некоторого метакласса. Речь идет о том, что в естественной речи мы часто опускаем слово "пример" или "экземпляр", говоря просто "класс". Так, фразу "Атрибут **возраст** класса **сотрудник** имеет значение **30 лет**" следует записать более точно: "Атрибут **возраст** экземпляра класса **сотрудник** имеет значение **30 лет**";
- в каждый момент времени используется только то значение слова, которое приписано имени соответствующей конструкции языка UML 2.0. Все дополнительные особенности семантики должны быть указаны явным образом, без каких бы то ни было неявных предположений;
- термины языка UML 2.0 могут включать только один из допустимых префиксов, таких как "под-", "супер-" или "мета-". При этом сам термин с префиксом записывается одним словом.

В дополнение к этому используются следующие правила выделения текста:

- если используются ссылки на конструкции языка UML 2.0, а не на их представления в метамодели, следует применять обычный текст, без какого бы то ни было выделения;
- имена метаклассов являются элементом нотации языка UML 2.0 и представляют собой существительные и, возможно, присоединенные к ним прилагательные. В этом случае имя метакласса на английском записывается одним словом с выделением каждой составной части имени заглавной буквой (например, `ModelElement`, `StructuralFeature`);
- имена метаассоциаций и ассоциаций классов записываются аналогичным образом (например, `ElementReference`);

- имена других элементов языка UML 2.0 также записываются одним словом, но должны начинаться с маленькой буквы (например, `ownedElement`, `allContents`);
- имена метаатрибутов, которые принимают булевы значения, всегда начинаются с префикса "is" (например, `isAbstract`);
- перечислимые типы всегда заканчиваются словом "Kind" (например, `AggregationKind`);
- при ссылках в тексте на метаклассы, метаассоциации, метаатрибуты и т. д. должны всегда использоваться в точности те их имена, которые указаны в модели;
- имена стандартных обозначений, прежде всего текстовых стереотипов, заключаются в угловые кавычки и начинаются со строчной буквы (например, `<<type>>`).

Для спецификации допустимых форматов синтаксиса выражений языка UML 2.0 в текстовой нотации используется вариант *Формы Бэкуса—Наура* (БНФ), для которой применяются следующие соглашения:

- все не терминалы записываются курсивом и заключаются между угловыми скобками (например, `<`*`не терминал`*`>`);
- все терминалы (ключевые слова, строки и т. д.) заключаются между одиночными кавычками (например, `'or'`);
- определения не терминальных правил продукции обозначаются с помощью оператора `::=`;
- повторение элемента обозначается с помощью звездочки, помещенной после этого элемента: `*`;
- альтернативный выбор в выражении разделяется символом `|` (например, `<альтернатива-A> | <альтернатива-B>`);
- элементы, которые являются необязательными, заключаются в квадратные скобки (например, `[<элемент-x>]`);
- когда необходимо сгруппировать элементы, они заключаются в простые скобки. Например, выражение `(<элемент-1> | <элемент-2>)*` означает последовательность из одного или более элементов, каждый из которых является `<элемент-1>` или `<элемент-2>`.

Рассмотренные выше правила написания текста имеют непосредственное отношение к англоязычным терминам языка UML 2.0. Поскольку вопросы локализации языка UML 2.0 до настоящего времени не нашли своего отражения в работе OMG, отечественным разработчикам придется самостоятельно

дополнять эти правила на случай использования в качестве естественного русского языка. В книге мы будем придерживаться двух дополнительных рекомендаций.

- При описании семантики языка UML 2.0 все имена его стандартных элементов (метаклассов, метаассоциаций, метаатрибутов) допускается записывать на русском с дополнительным указанием оригинального имени на английском языке. При этом, хотя имена стандартных элементов могут состоять из нескольких слов, согласно сложившейся отечественной традиции, будем их записывать раздельно с заглавной буквы (например, `Ассоциация класс`, `Элемент модели`, `Пространство имен`).
- При разработке конкретных моделей систем в форме диаграмм языка UML 2.0 целесообразно применять оригинальные англоязычные термины, придерживаясь описанных выше правил (кроме, возможно, пояснительного текста на русском). Причина этой рекомендации вполне очевидна — последующая инструментальная реализация модели может оказаться невозможной, если не следовать оригинальным правилам использования текста в языке UML 2.0. Это правило не распространяется на отдельные примеры и фрагменты диаграмм, которые приводятся в тексте книги с чисто иллюстративными целями и лишь раскрывают особенности использования стандартных элементов языка UML 2.0.

Приведенные дополнительные рекомендации не противоречат оригинальным правилам языка UML 2.0, а только уточняют рамки использования естественного языка при построении моделей и при описании самого языка. Поскольку описание семантики любого формального языка связано с проблемой его интерпретации, полностью обойтись без естественного языка не представляется возможным. Если вопросы использования оригинальных терминов при построении логических и физических моделей не вызывают сомнений у большинства программистов, то процесс построения концептуальных моделей сложных систем формализован в меньшей степени. Именно по этой причине исходные требования к системе формулируются на естественном для разработчиков языке (в нашем случае, на русском). В любом случае эти аспекты использования языков при построении моделей многогранны и могут служить предметом выбора самих разработчиков.

В рамках языка UML 2.0 все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название *диаграмм*. В терминах языка UML 2.0 определены следующие типы диаграмм:

- диаграммы структуры (Structure Diagrams):
  - диаграмма классов (Class diagram);
  - диаграмма композитной структуры (Composite Structure diagram);

- диаграмма пакетов (Package diagram);
- диаграмма объектов (Object Diagram);
- диаграмма компонентов (Component diagram);
- диаграмма развертывания (Deployment diagram);

❒ диаграммы поведения (Behavior Diagrams):

- диаграмма вариантов использования (Use Case diagram);
- диаграмма деятельности (Activity diagram);
- диаграмма конечного автомата (State Machine diagram);

❒ диаграммы взаимодействия (Interaction Diagrams):

- диаграмма последовательности (Sequence diagram);
- диаграмма коммуникации (Communication diagram);
- диаграмма обзора взаимодействия (Interaction Overview diagram);
- временная диаграмма (Timing diagram).

Диаграммы структуры или структурные диаграммы предназначены для представления статической структуры элементов в модели системы. На них изображаются те элементы модели и отношения, которые не зависят от времени. Элементы на диаграммах структуры могут включать абстрактные понятия, а также понятия предметной области и реализации системы. Например, диаграмма структуры для системы резервирования авиабилетов может включать классификаторы, которые представляют алгоритмы назначения мест, билеты и сервис авторизации кредита клиентов. На структурных диаграммах не изображаются детали динамического поведения, которые иллюстрируются диаграммами поведения. Однако на них могут быть показаны отношения с поведением классификаторов, изображенных на структурных диаграммах.

Диаграммы поведения изображают динамическое поведение объектов в системе, включая кооперации, деятельности, вызовы методов объектов и изменение их состояний. Динамическое поведение системы может быть описано в форме последовательности изменений, происходящих с элементами модели системы во времени. Диаграммы структуры и поведения в свою очередь состоят из диаграмм других типов, которые могут быть представлены в форме классификации (рис. 2.12). При этом все диаграммы нижнего уровня имеют специальное название — *канонических диаграмм*.

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML 2.0. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информа-

ция, которая необходима для реализации проекта системы произвольной сложности.

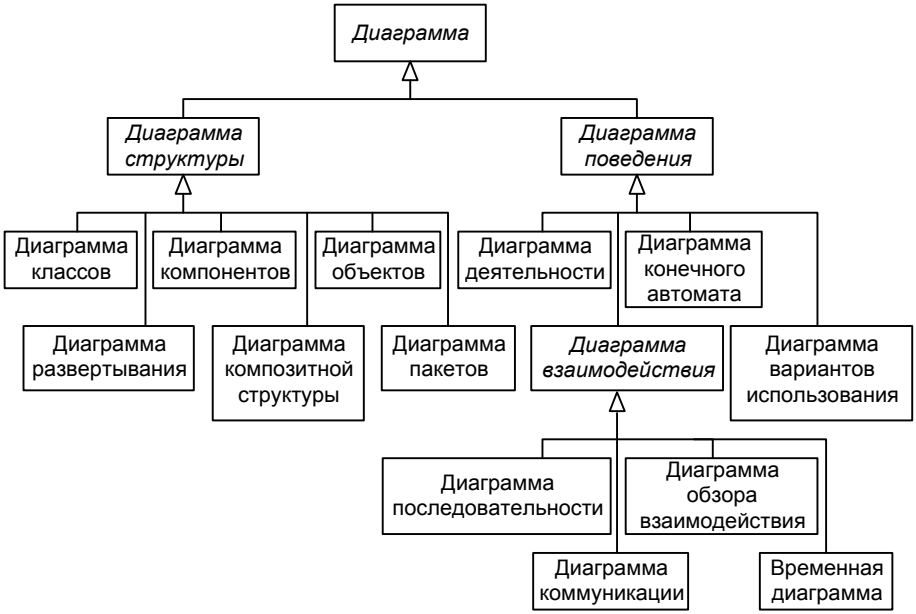


Рис. 2.12. Классификация канонических диаграмм языка UML 2.0

Следует заметить, что количество канонических диаграмм в языке UML 2.0 по сравнению с предыдущими версиями выросло до 13 (вместо имевших место ранее 8 диаграмм). Новым в языке UML 2.0 является также то, что допускается смешивать элементы различных канонических диаграмм в произвольных комбинациях. Тем самым на одной диаграмме могут быть изображены элементы структуры и элементы поведения. Так, например, конечный автомат или вариант использования может быть изображен вложенным внутрь некоторого класса. Хотя границы между различными типами диаграмм допускается не соблюдать строго, изучение языка UML 2.0 традиционно концентрируется на канонических диаграммах. Именно такой подход положен в основу структуры настоящей книги.

В качестве самостоятельных модельных представлений в языке UML 2.0 используются следующие канонические диаграммы:

- *диаграмма вариантов использования* (use case diagram) — диаграмма концептуального уровня, которая служит для представления актеров и вариантов использования системы, а также отношений между ними *(см. главу 3)*;

- *диаграмма классов* (class diagram) — диаграмма, которая служит для представления совокупности декларативных или статических элементов модели, таких как классы, типы, а также разнообразных отношений между ними *(см. главу 4)*;
- *диаграмма композитной структуры* (composite structure diagram) — диаграмма, которая изображает внутреннюю структуру классификаторов, таких как класс, компонент или кооперация, включая точки взаимодействия классификатора с другими частями системы *(см. главу 5)*;
- *диаграмма пакетов* (package diagram) — диаграмма логического уровня, которая служит для представления организации элементов модели по пакетам, а также зависимостей между пакетами, включая импорт пакетов и слияние пакетов *(см. главу 6)*;
- *диаграмма объектов* (object diagram) — диаграмма, которая служит для представления объектов и отношений между ними в конкретный момент времени. Она может рассматриваться как специальный случай диаграммы классов или диаграммы коммуникации *(см. главу 6)*;
- *диаграмма последовательности* (sequence diagram) — диаграмма, которая служит для представления взаимодействия элементов модели в форме последовательности сообщений и соответствующих событий на линиях жизни объектов *(см. главу 7)*;
- *диаграмма деятельности* (activity diagram) — диаграмма, которая изображает поведение объекта или системы с использованием моделей потока данных и потока управления *(см. главу 8)*;
- *диаграмма коммуникации* (communication diagram) — диаграмма, которая служит для представления взаимодействия между линиями жизни объектов в форме элементов внутренней структуры системы и передаваемых между ними сообщений *(см. главу 9)*;
- *диаграмма обзора взаимодействия* (interaction overview diagram) — вариант диаграммы деятельности, которая служит для представления только взаимодействия потока управления в некоторой агрегированной форме *(см. главу 9)*;
- *временная диаграмма* (timing diagram) — диаграмма взаимодействия, которая служит для представления изменения состояния или условий линии жизни отдельных экземпляров классификаторов во времени *(см. главу 9)*;
- *диаграмма конечного автомата* (state machine diagram) — диаграмма, которая служит для представления дискретного поведения, моделируемого посредством переходов в системах с конечным числом состояний. В частности, она может представлять последовательность состояний, через

которые объект или система проходят в течение жизни, реагируя на различные события *(см. главу 10)*;

- *диаграмма компонентов* (component diagram) — диаграмма физического уровня, которая служит для представления программных компонентов и зависимостей между ними *(см. главу 11)*;
- *диаграмма развертывания* (deployment diagram) — диаграмма физического уровня, которая служит для представления архитектуры распределенных программных систем. Она содержит информацию о размещении артефактов проекта системы по узлам, которые соединяются коммуникациями для создания сетевой инфраструктуры произвольной сложности *(см. главу 12)*.

Таким образом, интегрированная модель сложной системы в нотации UML 2.0 может быть представлена в виде совокупности указанных выше диаграмм (рис. 2.13).

Каждая из канонических диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML 2.0.

Рис. 2.13. Интегрированная модель сложной системы в нотации UML 2.0

2.6. Особенности изображения диаграмм в нотации UML 2.0

Большинство перечисленных диаграмм являются в своей основе графами специального вида, состоящими из вершин или узлов в форме различных геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, имеет в основном топологический характер, ни геометрические размеры, ни расположение элементов диаграмм (за исключением диаграммы последовательности и временной диаграммы) не имеют принципиального значения.

Для диаграмм языка UML 2.0 существуют три типа визуальных графических обозначений, которые важны с точки зрения заключенной в них информации.

- Геометрические фигуры на плоскости, играющие роль вершин графов соответствующих диаграмм. При этом сами геометрические фигуры выступают в роли *графических узлов* языка UML 2.0, а форма этих фигур (прямоугольник, эллипс) должна строго соответствовать изображению отдельных элементов языка UML 2.0 (класс, вариант использования, состояние, деятельность). Графические узлы диаграмм языка UML 2.0 имеют фиксированную семантику, переопределять которую пользователям не разрешается. Графические узлы должны иметь собственные имена, а возможно, и другой текст, который содержится внутри границ соответствующих геометрических фигур или, как исключение, вблизи этих фигур.
- Графические пути, которые представляются различными линиями на плоскости. Пути в языке UML 2.0 обобщают понятие дуг и ребер из теории графов, но имеют менее формальный характер и более развитую семантику.
- Специальные графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм и имеющие характер дополнительной спецификации (украшений).

На всех диаграммах нотации UML 2.0 в качестве узлов используются геометрические фигуры на плоскости. Однако некоторые фигуры, например кубы, могут представлять собой двумерные проекции трехмерных геометрических тел. В этом случае они рисуются как фигуры на плоскости.

Таким образом, в языке UML 2.0 используется четыре основных вида конструкций.

- Графические узлы на плоскости, которые изображаются с помощью геометрических фигур и могут иметь различную высоту и ширину с целью

размещения внутри этих фигур других конструкций языка UML 2.0. Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML 2.0. Информация, содержащаяся внутри узлов, имеет важное значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде.

- Пути, которые представляют собой последовательности из отрезков линий, соединяющих отдельные графические узлы. При этом концевые точки отрезков линий должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения узлов диаграмм, как это принято в теории графов. С концептуальной точки зрения путям в языке UML 2.0 придается особое значение, поскольку они являются простыми топологическими сущностями. С другой стороны, отдельные части пути или сегменты могут не существовать сами по себе вне содержащего их пути. Пути всегда соприкасаются с другими графическими символами на обеих границах соответствующих отрезков линий. Другими словами, пути не могут обрываться на диаграмме линией, которая не соприкасается ни с одним графическим узлом. Как отмечалось выше, пути могут иметь в качестве окончания или терминатора специальную графическую фигуру — значок, который изображается на одном их концов линий.
- Значки или пиктограммы. Значок представляет собой графическую фигуру фиксированного размера и формы. Она не может увеличивать свои размеры, чтобы разместить внутри себя дополнительные символы. Значки могут размещаться как внутри других графических конструкций, так и вне их. Примерами значков могут служить окончания путей между элементами диаграмм и некоторые другие дополнительные обозначения (украшения).
- Строки текста. Служат для представления различных видов информации в некоторой грамматической форме. Предполагается, что каждое использование строки текста должно соответствовать синтаксису соответствующих элементов нотации языка UML 2.0. Посредством этого синтаксиса может быть реализован грамматический разбор этой строки, который необходим для получения адекватной информации о модели. Например, строки текста в различных секциях обозначения класса могут соответствовать атрибутам этого класса или его операциям. При использовании строк текста необходимо выполнение важного условия — семантика всех допустимых символов должна быть заранее определена в языке UML 2.0 или служить предметом его расширения в конкретной модели.

В общем случае при графическом изображении диаграмм в нотации языка UML 2.0 необходимо придерживаться следующих основных рекомендаций.

- Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки отдельных диаграмм необходимо учитывать все сущности, важные с точки зрения контекста данной модели и типа диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.
- Все сущности на диаграмме модели должны быть одного концептуального уровня. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.
- Вся информация о сущностях должна быть явно представлена на диаграммах. Речь идет о том, что хотя в языке UML 2.0 при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), необходимо стремиться к явному указанию свойств всех элементов диаграмм.
- Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезных проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными свойствами в одном пространстве имен также приводит к их неоднозначной интерпретации и может служить источником проблем при реализации модели.

Наличие в инструментальных CASE-средствах встроенной поддержки визуализации различных диаграмм языка UML 2.0 позволяет в некоторой степени исключить ошибочное использование тех или иных графических символов, а также контролировать уникальность имен элементов диаграмм. Однако неформальный характер языка UML 2.0 может служить источником потенциальных ошибок, которые в полном объеме вряд ли будут выявлены инструментальными средствами. При этом необходимо помнить, что вся ответственность за окончательный контроль непротиворечивости модели лежит на разработчике. Именно это обстоятельство требует от всех разработчиков глубокого знания нотации и семантики всех элементов языка UML 2.0.

- Диаграммы не следует перегружать текстовой информацией. Принято считать, что визуализация модели является наиболее эффективной, если она содержит минимум пояснительного текста. Как правило, наличие больших фрагментов развернутого текста служит признаком недостаточной проработанности модели или ее неоднородности, когда в рамках одной модели представляется различная по характеру информация. Поскольку общая декомпозиция модели на отдельные типы диаграмм способна удовлетворить самые детальные представления разработчиков о системе, важно уметь правильно отображать те или иные сущности и аспекты моделирования в соответствующие элементы канонических диаграмм.
- Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментарии), не должны приниматься во внимание разработчиками. В то же время отдельные общие фрагменты могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML 2.0 представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.
- Количество диаграмм различных типов для модели конкретного приложения не является строго фиксированным. Речь идет о том, что для простых приложений нет необходимости строить все без исключения типы канонических диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и этот факт не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, которое будет выполняться локально на компьютере пользователя. Важно понимать, что состав диаграмм зависит от специфики конкретного проекта системы.
- Любая из моделей системы должна содержать только те элементы, которые определены в нотации языка UML 2.0. Имеется в виду требование начинать разработку проекта с использованием только тех конструкций, которые определены в метамодели UML 2.0. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только в случае отсутствия необходимых базовых элементов языка UML 2.0 следует использовать механизмы расширения для адекватного представления конкретной модели системы.

При этом не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML 2.0.

- Каждая диаграмма в нотации языка UML 2.0 имеет область содержания для изображения графических узлов и путей между ними, которые представляют собой собственно элементы модели в нотации UML 2.0. Дополнительно диаграмма может быть заключена во фрейм (прямоугольную рамку) и иметь заголовок, как это изображено на рис. 2.14.

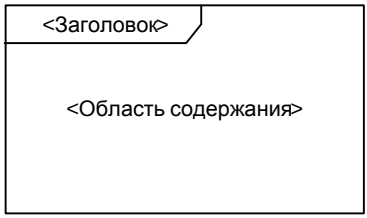


Рис. 2.14. Изображение диаграммы языка UML 2.0 в виде фрейма

- Фрейм используется в тех случаях, когда отдельные элементы диаграммы имеют графическую границу с другими элементами диаграммы. В общем случае фрейм может быть опущен. В этом случае предполагается, что он ограничивает границу области содержания диаграммы. В область содержания диаграммы помещают графические символы; при этом главные графические символы определяют тип диаграммы. Например, диаграмма классов является диаграммой, главными символами в области содержания которой являются символы классов.
- Заголовок диаграммы является строкой текста, записанной в прямоугольнике с обрезанным углом в верхнем левом углу фрейма и имеющей следующий синтаксис:

  [<*тип диаграммы*>]<*имя*>[<*параметры*>]

  Заголовок диаграммы представляет тип диаграммы, имя и параметры соответствующего пространства имен или элемента модели, который владеет элементами модели в области содержания диаграммы. Если фрейм диаграммы опущен, заголовок также опускается. Диаграммы UML 2.0 могут иметь следующие типы диаграмм как часть заголовка:

  - `activity` (для фреймов диаграммы деятельности);
  - `class` (для фреймов диаграммы классов);
  - `component` (для фреймов диаграммы компонентов);
  - `interaction` (для фреймов диаграмм взаимодействия);

- `package` (для фреймов диаграммы пакетов);
- `state machine` (для фреймов диаграммы конечного автомата);
- `use case` (для фреймов вариантов использования).

❒ Кроме полной формы типа диаграммы для заголовка также могут использоваться следующие сокращенные формы:

- `act` (для фреймов диаграммы деятельности);
- `cmp` (для фреймов диаграммы компонентов);
- `sd` (для фреймов диаграмм взаимодействия);
- `pkg` (для фреймов диаграммы пакетов);
- `stm` (для фреймов диаграммы конечных автоматов);
- `uc` (для фреймов диаграммы вариантов использования).

2.7. Механизмы расширения в языке UML 2.0

Хотя язык UML 2.0 обладает богатым множеством понятий и графических символов для моделирования обычных программных систем, на практике разработчики могут столкнуться с необходимостью включить в модель дополнительные свойства или элементы, которые в явном виде не определены в языке UML 2.0. При этом разработчикам может потребоваться включить в модель графическую информацию, такую, например, как дополнительные значки и символы. Порядок включения в модель элементов с уточненной или расширенной семантикой, а также правила модификации существующих элементов языка UML 2.0 для более точного отражения специфики моделируемых систем специфицируют так называемые *механизмы расширения*.

В языке UML 2.0 предусмотрены три механизма расширения, которые могут использоваться совместно или раздельно для определения новых элементов модели с дополнительной семантикой, нотацией и свойствами по сравнению со специфицированными в метамодели языка UML 2.0 элементами. Такими механизмами являются стереотип, ограничение и помеченное значение.

В общем случае механизмы расширения языка UML 2.0 предназначены для выполнения следующих задач:

❒ уточнения или специализации общих элементов метамодели при разработке конкретных моделей в нотации языка UML 2.0;

❒ определения таких расширений языка UML 2.0, которые зависят от специфики моделируемого процесса или от языка реализации программного кода;

- присоединения произвольной семантической или несемантической информации к элементам модели.

Хотя вопросы расширения метамодели собственно UML 2.0 выходят за пределы настоящей книги, разработчикам следует знать о потенциальной возможности явного добавления в язык UML 2.0 новых метаклассов и других конструкций языка. Однако при этом необходимо соблюдать правила порождения новых метаклассов от уже имеющихся в языке UML 2.0.

2.7.1. Стереотип

Наиболее важным из механизмов расширения является механизм стереотипов, на основе которого становится возможным расширение семантики существующих метаклассов и других конструкций языка UML 2.0.

Стереотип (stereotype) — элемент модели, который расширяет семантику базового элемента метамодели языка UML 2.0.

В моделях языка UML 2.0 стереотипы могут быть представлены следующими способами:

- в форме текста, заключенного в двойные угловые кавычки и размещенного выше или перед именем элемента модели (см. рис. 2.6 и 2.12). Первая буква текста имени стереотипа не должна быть прописной буквой. Если применяются несколько стереотипов, то их имена изображаются в форме разделенного запятыми списка с парой кавычек. Если расширяемый элемент модели имеет некоторое ключевое слово, то имя стереотипа должно изображаться вблизи этого ключевого слова в отдельных кавычках. Важно помнить, что имена текстовых стереотипов не должны совпадать с именами ключевых слов языка UML 2.0 для расширяемого элемента модели;
- в форме графической пиктограммы, которая заменяет текст имени соответствующего стереотипа. Каждый элемент модели, который имеет некоторое графическое представление, может иметь дополнительные пиктограммы стереотипов. При этом интерпретация различных пиктограмм стереотипов является так называемой точкой вариации или изменения семантики, которая должна быть специфицирована самим разработчиком или используемым CASE-средством;
- в форме прямоугольника класса с уменьшенной пиктограммой стереотипа внутри этого прямоугольника, расположенного, как правило, в верхнем правом углу. Также как и для пиктограмм стереотипов, интерпретация

уменьшенных пиктограмм стереотипов является точкой вариации семантики, которая должна быть специфицирована самим разработчиком или используемым CASE-средством.

Некоторые CASE-средства могут использовать собственные изображения для пиктограмм стереотипов, которые не противоречат определению данного механизма расширения. Отдельные элементы языка UML 2.0 уже используют пиктограммы стереотипов для своего представления. Типичным примером является рассматриваемое в *главе 3* понятие актера, для которого в качестве основного графического символа используется пиктограмма "человечка". В случае, когда элемент модели расширяется посредством стереотипа в форме некоторой пиктограммы, эта пиктограмма стереотипа заменяет на диаграммах представление соответствующего графического элемента по умолчанию.

Стереотипы обеспечивают применение специальной терминологии предметной области в дополнение или вместо терминологии, используемой для расширяемого элемента в метамодели языка UML 2.0. Стереотипы также предоставляют некоторый способ классификации элементов модели и возможность добавления в язык UML 2.0 "виртуальных" метаклассов с новыми свойствами и семантикой.

Важно помнить, что стереотипы предназначены для расширения именно семантики, но не структуры уже описанных метаклассов. При определении новых стереотипов разработчики также должны знать те стереотипы, которые определены в языке UML, чтобы исключить возможные проблемы с их интерпретацией.

Другие встроенные механизмы расширения языка UML 2.0 основываются на понятии списка свойств матеклассов, который может содержать ограничения и помеченные значения. Эти механизмы позволяют разработчикам включать дополнительные свойства и семантику в те или иные элементы разрабатываемых ими моделей.

2.7.2. Ограничение

Ограничение (constraint) — логическое условие, выраженное текстом на естественном языке или на языке, который может быть прочитан некоторой машиной, с целью декларирования семантики некоторого элемента модели.

Ограничение является утверждением, которое указывает на то, что при правильном проектировании системы должно быть удовлетворено некоторое

дополнительное условие. Ограниченные элементы представляют собой такие элементы, для которых требуется оценивать спецификацию ограничения. При этом ограничение представляет собой дополнительную семантическую информацию, присоединенную к ограниченным элементам. Некоторые ограничения предопределены в языке UML (такое как ограничение для ассоциации "xor"), другие могут быть специфицированы самим разработчиком.

Ограничение изображается в форме строки текста в фигурных скобках {} в соответствии со следующей БНФ:

<*ограничение*> ::= '{' [<*имя*> ':'] <*логическое выражение*> '}'

Имя ограничения является необязательным, поскольку в большинстве случаев ограничения являются безымянными. *Логическое выражение*, которое при отсутствии имени часто называют собственно ограничением, может быть представлено в форме текста на естественном языке, на некотором языке программирования (например, Java) или на специально разработанном для этой цели языке — *языке объектных ограничений* (Object Constraint Language, OCL). Последний специально служит для корректной записи ограничений, в том числе и для записи ограничений и правил правильного построения выражений языка UML 2.0.

Следует помнить, что логическое условие ограничения должно принимать значение "истина" для всех возможных реализаций соответствующей модели. Это обстоятельство делает исключительно важным корректное определение ограничений всеми разработчиками.

Каждое ограничение может относиться или ссылается на один или несколько элементов модели. Оценивание ограничения выполняется в некотором контексте, являющимся пространством имен, к которому относятся ограниченные элементы. Для того чтобы ограничение было удовлетворено, логическое условие при оценивании должно быть истинным. Оценивание логического условия для ограничения не должно иметь побочных эффектов. Ограничение не может быть применено к самому себе.

Для элемента, нотацией которого является строка текста (такого, как атрибут класса), строка ограничения может следовать за строкой текста элемента в скобках. Для ограничения, которое применяется к отдельному элементу (такому как класс или ассоциация), строка ограничения может быть помещена рядом с символом этого элемента, предпочтительно вблизи имени, если таковое имеется. Строка ограничения может быть помещена в символе примечания и присоединена к каждому из символов ограниченных элементов посредством пунктирной линии. Для ограничения, которое применяется

к двум элементам (таким как два класса или две ассоциации), ограничение может быть изображено в форме пунктирной линии между этими элементами, помеченной строкой ограничения в скобках.

При спецификации ограничения дополнительно может быть указан контекст ограничения, который может быть использован в качестве пространства имен для интерпретации имен, используемых в этой спецификации. Например, для того чтобы ссылаться на контекст элемента, в языке OCL существует ключевое слово "self". Если используется формальный язык, такой как OCL, то CASE-средства могут обладать способностью проверять синтаксис логических условий ограничений.

Если ограничение изображается в форме пунктирной линии между двумя элементами, то на одном ее конце может быть помещена стрелка. Направление этой стрелки является релевантной информацией для ограничения. Элемент на стрелке отображается в первой позиции, а элемент на противоположном конце стрелки отображается во второй позиции в коллекции ограниченных элементов. Для 3-х или более путей одного и того же вида (таких как пути обобщений или пути ассоциаций) ограничение может быть присоединено к пересечению пунктирных линий всех путей.

2.7.3. Помеченное значение

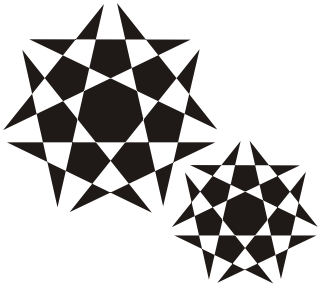
Помеченное значение (*tagged value*) — явное определение некоторого свойства объекта в форме пары "имя — значение". В помеченном значении имя называется также *тегом* (*tag*).

Отдельные теги предопределены в языке UML, другие могут быть определены самим разработчиком. Помеченное значение в языке UML 2.0 представляется только в форме текста, заключенного в фигурные скобки и размещенного ниже или после имени элемента модели. При этом определяющим символом помеченного значения является знак равенства, который отделяет тег от его значения. Помеченное значение также может быть изображено в форме примечания, присоединенного к соответствующему элементу модели.

В заключение этой главы следует отметить, что процесс построения отдельных канонических диаграмм при разработке моделей программных систем имеет свои особенности, которые не описаны в метамодели языка UML 2.0. Хотя для этой цели в общем случае может использоваться различная методология, наиболее естественным в контексте языка UML является процесс ООАП, который получил специальное название — *рациональный унифицированный процесс* (Rational Unified Process, RUP). Концепция RUP и основные его элементы разработаны А. Джекобсоном в ходе его работы над языком UML.

Строго говоря, существующая на момент написания книги версия RUP определяет последовательность разработки канонических диаграмм языка UML 1.*x*. Тем не менее большинство рекомендаций RUP оказываются справедливыми и для разработки канонических диаграмм языка UML 2.0, если принять во внимание связь отдельных типов этих диаграмм с диаграммами языка UML 1.*x* предыдущих версий.

Кратко суть концепции RUP заключается в последовательной декомпозиции или разбиении процесса ООАП на отдельные фазы или этапы, на каждом из которых осуществляется разработка соответствующих типов канонических диаграмм для рассматриваемой модели системы. При этом на начальных этапах RUP строятся логические представления статической модели структуры системы, затем — логические представления модели поведения, а после этого — физические представления модели системы. В результате выполнения RUP должны быть построены все необходимые модели в форме тех или иных канонических диаграмм на языке UML 2.0. Тем самым порядок изложения канонических диаграмм в *части II* книги согласуется с общими рекомендациями рационального унифицированного процесса.



Часть II

Диаграммы визуального моделирования языка UML 2.0

Данная часть книги посвящена систематическому изучению канонических диаграмм, которые являются основой всех модельных представлений в языке UML 2.0. Построение визуальных моделей на практике предполагает использование некоторой методологии разработки программных систем, которая определяет последовательность разработки отдельных диаграмм. При этом большинство методологий рекомендуют начинать разработку моделей в нотации UML с диаграммы вариантов использования, которая стала своеобразной визитной карточкой языка UML. Наше рассмотрение также начинается с диаграммы вариантов использования, которая служит концептуальной основой всех других модельных представлений в нотации UML 2.0.

Глава 3

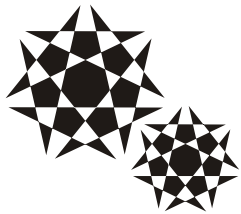


Диаграмма вариантов использования (use case diagram)

Процесс разработки визуальных моделей в нотации языка UML 2.0 обычно представляется в форме последовательной детализации модельных представлений от концептуальной модели к логической, а затем и к физической модели рассматриваемой программной системы. При этом исходной моделью, с которой начинается процесс моделирования в нотации UML, является модель или диаграмма вариантов использования. Она описывает функциональное назначение системы в самом общем виде с точки зрения всех ее пользователей и заинтересованных лиц. Говоря об особенностях этой канонической диаграммы, иногда отмечают, что соответствующая модель должна отвечать на вопрос о том, что должна делать система в процессе своего функционирования, не затрагивая вопрос о том, как она должна это делать.

3.1. Диаграмма вариантов использования — исходная концептуальная модель проектируемой системы

Диаграмма вариантов использования (use case diagram) — диаграмма, на которой изображаются варианты использования проектируемой системы, заключенные в границу субъекта, и внешние актеры, а также определенные отношения между актерами и вариантами использования.

3.1.1. Назначение диаграммы вариантов использования

Диаграмма вариантов использования предназначена для достижения следующих целей:

- определить общие границы функциональности проектируемой системы в контексте моделируемой предметной области;
- специфицировать требования к функциональному поведению проектируемой системы в форме вариантов использования;
- разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;
- подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

При разработке диаграммы вариантов использования нужно учитывать следующие особенности ее модельного представления. Прежде всего, функциональность рассматриваемой системы представляется в форме так называемых вариантов использования, с которыми взаимодействуют некоторые внешние сущности или актеры. В качестве актеров могут выступать любые объекты, субъекты или системы, взаимодействующие с моделируемой системой извне. В свою очередь варианты использования служат для описания сервисов, которые система предоставляет различным актерам. Совокупность всех вариантов использования, рассматриваемых в контексте поведения проектируемой системы, заключается в границу описываемой системы или образует ее субъект. При этом в рамках данной диаграммы не делается никаких предположений относительно того, как технически или физически будет реализовано выполнение вариантов использования и взаимодействие актеров с системой.

Рассматривая диаграмму вариантов использования в качестве исходного концептуального представления проектируемой системы, ее можно сопоставить с моделью системы в форме "черного ящика" (рис. 3.1).

Визуально диаграмма вариантов использования представляет собой граф специального вида, который содержит специальные условные изображения вариантов использования, актеров и отношений между ними. Все варианты использования на отдельной диаграмме заключают в прямоугольник, который обозначает границу проектируемой системы или субъект. При этом данный граф может содержать только определенные типы взаимосвязей между актерами и вариантами использования.

В настоящее время среди разработчиков принято считать, что основное назначение диаграммы вариантов использования заключается в спецификации

функциональных требований к проектируемой системе. Поскольку требования выступают в качестве исходных данных для разработки системы, визуализация их в форме диаграммы вариантов использования не только повышает наглядность представления, но и позволяет целенаправленно управлять процессом разработки других моделей в нотации UML.

Рис. 3.1. Проектируемая система и ее окружение

После построения диаграммы вариантов использования в нотации UML 2.0 разработчики получают в свое распоряжение не только унифицированную графическую модель для представления функциональности проектируемой системы, но и практическое средство для итеративного уточнения требований к проектируемой системе. При этом уточнение требований происходит в форме их периодического обсуждения и согласования со всеми заинтересованными лицами в ходе выполнения проекта, а степень реализации требований служит одним из показателей завершенности проекта в целом. С формальной точки зрения диаграмма вариантов использования является специализацией диаграммы классов, для которой изображаемые классификаторы ограничиваются только актерами и вариантами использования.

3.1.2. Субъект вариантов использования

В языке UML 2.0 диаграмма вариантов использования является моделью поведения, которым могут обладать различные элементы модели проектируемой системы.

Субъектом (subject) в контексте языка UML 2.0 называется любой элемент модели, который обладает функциональным поведением.

Субъект вариантов использования может быть некоторой физической системой или любым другим элементом с поведением, таким как подсистема,

класс или компонент. Графически субъект на диаграмме вариантов использования изображается в форме прямоугольника, внутри которого размещаются отдельные варианты использования. Каждый субъект имеет имя, которое должно соответствовать имени элемента моделируемой системы, владеющего данным поведением.

Изображение субъекта в форме прямоугольника выбрано не случайно, поскольку этот символ является наиболее общим графическим представлением классификатора, частным случаем или специализацией которого является субъект. При этом изображение субъекта на диаграмме вариантов использования не является обязательным. Если его изображение отсутствует, то по умолчанию предполагается, что соответствующие варианты использования относятся к системе в целом.

Понятие субъекта не следует смешивать с рассматриваемым далее понятием актера. Эти понятия, как будет видно из дальнейшего изложения, являются различными модельными элементами в нотации UML 2.0.

Если проектируемая система является достаточно сложной или имеет большие масштабы, то для упрощения ее модельного представления она может быть декомпозирована или разбита на отдельные подсистемы. При таком разбиении каждая из подсистем рассматривается как изолированная или обособленная часть исходной системы. В этом случае для каждой из подсистем целесообразно разрабатывать отдельную диаграмму вариантов использования. В языке UML 2.0 предполагается, что субъект или его части реализуют все варианты использования, которые применяются в контексте этого субъекта. Однако необязательно, чтобы все варианты использования относились к некоторому субъекту. Субъект может, но не обязан, владеть вариантами использования, которые содержатся в нем.

3.2. Основные графические элементы диаграммы вариантов использования

Основными графическими элементами диаграммы вариантов использования являются варианты использования и актеры. В дополнение к ним на диаграмме вариантов использования могут изображаться комментарии. С этих понятий мы и приступим к изучению особенностей данной канонической диаграммы.

3.2.1. Вариант использования

Вариант использования (use case) представляет собой общую спецификацию совокупности выполняемых системой действий с целью предоставления некоторого наблюдаемого результата, который имеет значение для одного или нескольких актеров.

Другими словами, вариант использования представляет собой законченный фрагмент поведения системы с точки зрения тех или иных заинтересованных лиц. Хотя каждый вариант использования предполагает реализацию системой некоторой последовательности действий, которые должны быть выполнены с целью предоставления соответствующего результата или сервиса, сами эти действия на диаграмме вариантов использования никак не изображаются.

В литературе можно встретить различные переводы термина use case, например — прецедент, функция, транслитерация *юзкейс* или просто без перевода — use case. Увы, здесь мы сталкиваемся с ситуацией, когда русскоязычная терминология языка UML оказывается далекой от унификации. Не углубляясь в лингвистический анализ корректности перевода отдельных терминов, автор оставляет за читателем право выбора наиболее удобного для него варианта перевода. Что касается автора, то, как можно видеть, он свой выбор уже сделал.

Цель спецификации отдельного варианта использования заключается в том, чтобы зафиксировать некоторый функциональный аспект или фрагмент поведения проектируемой системы без указания технических или физических особенностей его реализации. При этом каждый вариант использования является объявлением некоторого предполагаемого поведения проектируемой системы или отдельной ее части без ссылки на внутреннюю структуру этой системы или части. В этом смысле каждый вариант использования удобно рассматривать как отдельный сервис, который субъект предоставляет по запросу тех или иных заинтересованных лиц или актеров. Каждый такой сервис определяет один из возможных способов применения системы по своему целевому назначению.

Формально термин "вариант использования" относится к типу варианта использования. В общем случае каждый вариант использования специфицирует типовой фрагмент полезной функциональности, экземпляр которого соответствующий субъект предоставляет по требованию отдельного актера или пользователя. Тем самым предполагается, что эта функциональность всегда инициируется актером. Каждый вариант использования специфицирует единицу полезной функциональности, которую субъект предоставляет своим

пользователям. Эта функциональность всегда должна быть завершена для того, чтобы экземпляр варианта использования считался законченным. Формально экземпляр варианта использования считается законченным, если после его выполнения субъект или система перейдет в состояние, в котором не предполагается выполнение каких-либо дополнительных действий. В отдельных случаях вариант использования может включать возможные варианты своего основного поведения, в том числе исключительное поведение и обработку ошибок.

Общепринятым графическим изображением варианта использования в нотации языка UML 2.0 является эллипс, внутри которого или ниже его записывается *имя* (name) в форме строки текста. Имя используется для идентификации отдельного варианта использования, как элемента модели. Каждый вариант использования обязательно должен иметь некоторое имя, которое рекомендуется формулировать в форме законченного предложения, начинающегося, как правило, с существительного (рис. 3.2, *а*) или глагола (рис. 3.2, *б*). Сам текст имени варианта использования должен начинаться с заглавной буквы.

Рис. 3.2. Стандартное графическое обозначение варианта использования: *а* — предложение начинается с существительного; *б* — предложение начинается с глагола

Если на диаграмме вариантов использования изображается субъект системы, то эллипсы вариантов использования визуально помещаются внутри прямоугольника субъекта. Примерами вариантов использования могут являться следующие фрагменты законченного функционального поведения системы: получение справки о состоянии счета клиента, оформление заказа на покупку товара, идентификация пользователя при входе в систему, отображение графической формы на экране монитора и другие поведения рассматриваемых систем.

Поскольку в языке UML 2.0 вариант использования является специализацией классификатора, для его изображения на диаграмме может быть также использован символ стандартной нотации классификатора в форме прямоугольника с необязательными секциями для его характеристик. Если данный графический символ применяется для представления варианта использования,

то он должен содержать специальное ключевое слово <<use case>>, которое размещается выше имени варианта использования в форме стереотипа (рис. 3.3, *а*). На диаграммах UML 2.0 допускается также изображать вариант использования символом классификатора, который вместо ключевого слова содержит декоративную пиктограмму в форме небольшого эллипса в правом верхнем углу прямоугольника (рис. 3.3, *б*).

Рис. 3.3. Графическое обозначение варианта использования в форме символа классификатора: *а* — с ключевым словом <<use case>>; *б* — с пиктограммой эллипса

Любая диаграмма вариантов использования всегда должна содержать конечное множество вариантов использования, которые в совокупности должны специфицировать все предполагаемые аспекты ожидаемого поведения системы. Для удобства множество вариантов использования можно рассматривать как отдельный пакет. В зависимости от ситуации поведение варианта использования может быть дополнительно описано спецификацией в форме модели взаимодействия, деятельности или конечного автомата, с помощью предусловий и постусловий, а также текстом на естественном языке в форме сценариев. Все эти описания в последующем должны быть объединены в общей модели системы.

3.2.2. Актер

Актер (actor) представляет собой любую внешнюю по отношению к проектируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач.

Формально в контексте языка UML 2.0 каждый актер специфицирует некоторую роль, которую играет пользователь или любая другая система, взаимодействующая с субъектом. Термин "роль" здесь используется неформально.

В модели вариантов использования пользователи и любые другие заинтересованные лица, которые могут взаимодействовать с субъектом, представляются как актеры. Актеры всегда моделируют сущности, которые находятся

за пределами границ системы. Если некоторая внешняя сущность выступает в нескольких ролях, то она должна быть изображена на диаграмме в виде нескольких актеров.

В литературе можно встретить различные переводы термина actor, например — пользователь, субъект, роль, актант, или транслитерация *эктор*. Некоторые из них, как, например, субъект, могут внести путаницу в понимание диаграммы. Другие, как, например, эктор, напоминают узко специальный жаргон. Хотя автор оставляет право выбора за читателем, но упомянуть о возможных несогласованностях терминологии все же он счел возможным.

Общепринятым графическим обозначением актера на диаграммах в нотации языка UML 2.0 является фигурка "проволочного человечка", под которой записывается обязательное имя актера (рис. 3.4, *а*). Поскольку актер является специализацией классификатора, он может изображаться также символом прямоугольника с ключевым словом `<<actor>>`, записанным в форме стереотипа выше его имени (рис. 3.4, *б*). Дополнительно в языке UML 2.0 допускается изображать актера в форме предложенной разработчиком пиктограммы, внешний вид которой, по мнению самого разработчика, более наглядно иллюстрирует характер выполняемой им роли (рис. 3.4, *в*).

Рис. 3.4. Варианты символов графического обозначения актера: *а* — в виде "проволочного человечка"; *б* — в виде прямоугольника; *в* — в форме пиктограммы

Имена актеров должны начинаться с заглавной буквы и удовлетворять общим требованиям использования имен для типов и классов модели. Следуя определению актера, в качестве его имени нужно всегда использовать существительное, возможно, с пояснительными словами. Как правило, имя актера записывается ниже графического изображения актера либо внутри символа классификатора. Имена абстрактных актеров должны быть записаны курсивом.

Актеры по своему определению служат для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют

с системой и используют ее в качестве отдельных категорий пользователей. Поэтому в качестве актеров могут выступать другие системы, в том числе подсистемы проектируемой системы или ее отдельные части, такие как классы или компоненты. Примерами актеров могут быть: кассир, клиент банка, банковский служащий, президент, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон, налоговая инспекция и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

Следует заметить, что актер не обязательно представляет ту или иную физическую сущность в целом, но только лишь отдельный аспект или роль этой сущности при взаимодействии с субъектом. Таким образом, отдельный физический экземпляр сущности может играть несколько ролей в форме нескольких актеров. И, наоборот, один актер может соответствовать нескольким физическим сущностям. Важно понимать, что наличие актера на диаграмме вариантов использования предполагает спецификацию некоторого согласованного множества ролей, в рамках которого могут выступать пользователи данной системы в процессе взаимодействия с ней. Неявно предполагается, что при взаимодействии с системой конкретного актера он играет или выступает в одной своей роли.

Любая конкретная физическая сущность, которая согласуется с определением актера, представляет собой экземпляр или пример актера. Для моделируемой программной системы актерами могут быть как непосредственные пользователи, так и другие программные, организационные, информационные или аппаратные системы. Поскольку пользователи проектируемой системы всегда являются внешними по отношению к ней, то они всегда представляются в виде актеров. Наглядный пример актера — посетитель интернет-магазина, имеющий свои собственные параметры аутентификации, предпочтения и интересы. Однако множество его ролей является вполне фиксированным — это выбор и покупка представленных в магазине товаров.

Поскольку актер всегда находится вне проектируемой системы, его внутренняя структура в модели никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Другими словами, на диаграмме вариантов использования невозможно что-либо утверждать о внутренней структуре и поведении актера, кроме его коммуникации с субъектом. На диаграмме вариантов использования эта коммуникация может быть выражена только посредством ассоциаций между актерами и вариантами использования.

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым обра-

зом. Такая общность свойств и поведения представляется в виде рассматриваемого далее отношения обобщения, которое моделирует соответствующую общность ролей. Совокупность всех отношений, которые могут присутствовать на диаграмме вариантов использования, будет рассмотрена далее в этой главе.

3.2.3. Комментарий

Комментарий (comment) в языке UML 2.0 предназначен для включения в модель произвольной текстовой информации в форме примечания, которое может быть присоединено к одному или нескольким элементам разрабатываемой модели. В качестве такой информации могут быть, например, пояснения разработчика относительно назначения элементов диаграммы, рекомендации руководителя проекта по реализации модели, справочная информация об авторе и особенностях разработки отдельных элементов модели. Применительно к диаграммам вариантов использования комментарий может иметь уточняющую информацию, относящуюся к контексту тех или иных вариантов использования и актеров.

Графически комментарий на всех типах диаграмм изображается в форме прямоугольника с "загнутым" правым верхним уголком (рис. 3.5, *а*). Этот символ имеет свое собственное название — *примечание* (note), внутри которого содержится тело или текст комментария. Комментарий может относиться к любому элементу диаграммы вариантов использования, который при этом называется аннотируемым элементом. При этом символ примечания должен быть соединен с аннотируемым элементом пунктирной линией (рис. 3.5, *б*). Считается, что комментарий находится в собственности аннотируемого элемента.

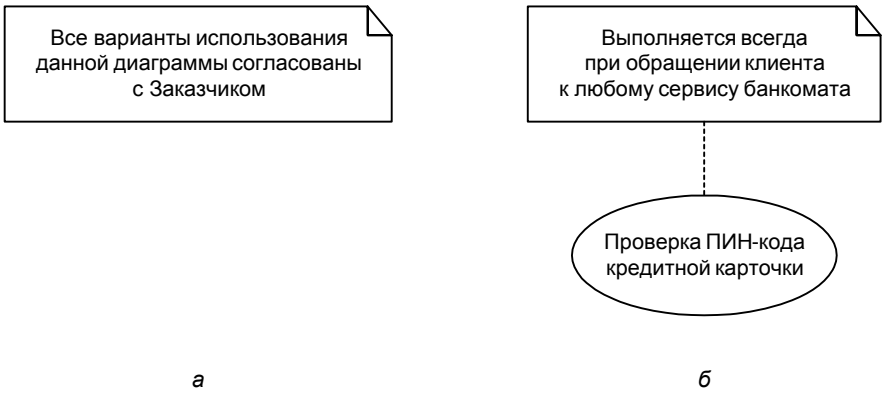


Рис. 3.5. Примеры комментариев в языке UML 2.0: *а* — в виде отдельного прямоугольника; *б* — в виде прямоугольника, присоединенного к аннотируемому элементу

Если комментарий относится к нескольким элементам, то от него проводятся, соответственно, несколько пунктирных линий к каждому из аннотируемых элементов. Пунктирная линия, которая соединяет символ примечания с аннотируемым элементом или элементами, может быть не указана в том случае, если из контекста ясно, к какому элементу модели относится комментарий, или это не важно для рассматриваемой диаграммы.

Комментарии могут присутствовать не только на диаграмме вариантов использования, но и на других типах диаграмм языка UML 2.0. Дополнительно символ примечания в языке UML 2.0 может применяться для размещения в нем ограничений и помеченных значений соответствующих элементов модели. Однако комментарий, в отличие от ограничений и помеченных значений, не добавляет никакой семантики к аннотируемым элементам. Поскольку ограничения могут достаточно жестко регламентировать различные аспекты реализации системы, на диаграммах вариантов использования размещать их не рекомендуется. По этой причине ограничения и помеченные значения более подробно будут рассмотрены далее применительно к другим типам канонических диаграмм языка UML 2.0.

3.3. Отношения на диаграмме вариантов использования

Между элементами диаграммы вариантов использования могут существовать различные взаимосвязи или отношения.

Под *отношением (relationship)* в языке UML 2.0 понимается произвольная семантическая взаимосвязь между отдельными элементами модели.

В общем случае отношение является абстрактным понятием (метаклассом) и ссылается на один или более связанных с ним элементов модели. Отношение не имеет специальной семантики и общей нотации. Различные подклассы отношения имеют собственную семантику и графическую нотацию, которая в большинстве случаев является вариацией линии, проведенной между связанными элементами.

На диаграмме вариантов использования отношения могут связывать актеров с вариантами использования, а также актеров и варианты использования между собой. В языке UML 2.0 имеется несколько стандартных видов отношений, которые могут быть изображены на диаграммах вариантов использования. Это отношения ассоциации, обобщения, включения и расширения. Именно эти виды отношений рассматриваются далее в этой главе.

3.3.1. Отношение ассоциации

Ассоциация (association) является одним из фундаментальных понятий в языке UML 2.0 и может использоваться на различных канонических диаграммах при построении визуальных моделей. Применительно к диаграммам вариантов использования отношение ассоциации может служить только для обозначения взаимодействия актера с вариантом использования. Другими словами, на диаграмме вариантов использования ассоциация всегда является бинарной и специфицирует семантические особенности отдельного взаимодействия актера и варианта использования.

На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией, соединяющей актера и вариант использования. При этом следует различать направленные и ненаправленные ассоциации. Если направление взаимодействия актера и вариантом использования для разработчика не имеет принципиального значения, то такое отношение может быть представлено в форме ненаправленной ассоциации (рис. 3.6).

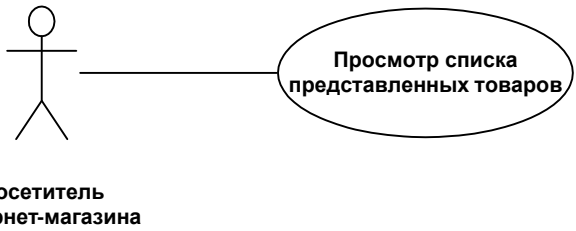


Рис. 3.6. Пример графического представления отношения ненаправленной ассоциации между актером и вариантом использования

Если же, по мнению разработчика, направление взаимодействия между актером и вариантом использования имеет значение, то такое отношение может быть представлено в форме направленной ассоциации (рис. 3.7). При этом направление ассоциации указывается в форме простой стрелки в форме буквы "V".

В контексте диаграммы вариантов использования направленная ассоциация от актера к варианту использования может указывать на тот факт, что данный актер является инициатором выполнения соответствующего варианта использования. Так, например, посетитель интернет-магазина является инициатором выполнения варианта использования — просмотра списка представ-

ленных в интернет-магазине товаров. В этом случае такого актера часто называют *главным* или *основным актером*.

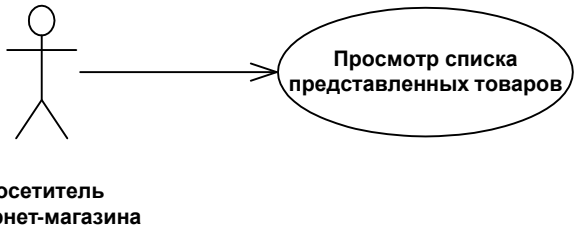


Рис. 3.7. Пример графического представления отношения направленной ассоциации между актером и вариантом использования

Направленная ассоциация от варианта использования к актеру может указывать также на тот факт, что данному актеру предоставляется справочная или отчетная информация о результатах функционирования моделируемой системы. Если при этом сами актеры не инициируют выполнения соответствующих вариантов использования, то таких актеров часто называют *второстепенными*.

Один вариант использования может иметь несколько ассоциаций с различными актерами, каждая из которых описывает, как экземпляр классификатора, реализующий этот вариант использования, взаимодействует с пользователем, играющим одну из ролей. В то же время два варианта использования, относящихся к одному и тому же субъекту, не могут быть ассоциированы, поскольку каждый из них индивидуально описывает законченный фрагмент функциональности субъекта.

Способ, которым несколько актеров взаимодействуют с отдельным вариантом использования, зависит от рассматриваемой ситуации и требует от разработчика дополнительной спецификации в модели. Например, некоторый вариант использования может требовать одновременного или параллельного действия двух отдельных актеров, как, например, при запуске ядерной ракеты. В других случаях вариант использования может требовать дополняющих и последующих действий актеров, когда, например, один актер начинает что-либо, а другой актер завершает это.

В общем случае отношение ассоциации может иметь собственное имя, а концевые точки ассоциации — имя и кратность. Однако эти характеристики ассоциации на диаграммах вариантов использования практически не специфицируются. Более детальное описание семантических особенностей ассоциации будет дано при рассмотрении диаграммы классов в *главе 4*.

3.3.2. Отношение включения

Отношение включения между двумя вариантами использования в языке UML 2.0 является частным случаем общего отношения зависимости.

Отношение *зависимости (dependency)* определяется как форма взаимосвязи между двумя элементами модели, предназначенная для спецификации того обстоятельства, что изменение одного элемента модели приводит к изменению некоторого другого элемента.

В общем случае зависимость является направленным бинарным отношением, которое связывает между собой только два элемента модели — независимый и зависимый.

Отношение *включения (include)* специфицирует тот факт, что некоторый вариант использования содержит поведение, определенное в другом варианте использования.

Семантически данное отношение означает, что поведение или выполнение одного варианта использования включается в качестве составного фрагмента в поведение или выполнение другого варианта использования. Отношение включения является направленным бинарным отношением в том смысле, что два варианта использования всегда упорядочены в отношении включения. Отношение включения может связывать в модели только два варианта использования.

Графически данное отношение обозначается как отношение зависимости в форме пунктирной линии с "V"-образной стрелкой, направленной от зависимого варианта использования к независимому варианту использования. Зависимый вариант использования часто называют также *базовым* или *включающим*, а независимый — *включаемым* вариантом использования. При этом линия со стрелкой обязательно помечается ключевым словом `<<include>>`, записанным в форме стереотипа (рис. 3.8).

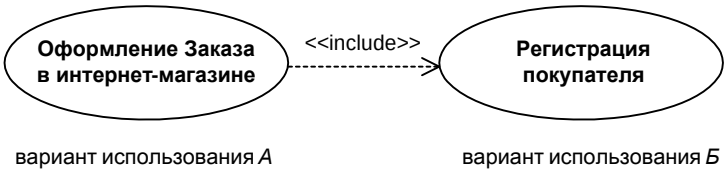


Рис. 3.8. Пример графического изображения отношения включения между вариантами использования

Например, отношение включения, направленное от варианта использования "Оформление Заказа в интернет-магазине" к варианту использования "Регистрация покупателя" (рис. 3.8), означает, что каждое выполнение варианта использования *А* всегда будет включать в себя функциональное поведение или выполнение варианта использования *Б*. В этом смысле поведение варианта использования *Б* является частью поведения варианта использования *А* на рассматриваемой диаграмме вариантов использования.

Поведение включаемого варианта использования всегда вставляется в поведение включающего варианта использования. Поэтому выполнение последовательности действий включаемого варианта использования происходит всегда при инициировании базового варианта использования, без проверки каких бы то ни было дополнительных условий. Однако поведение включающего варианта использования может зависеть от результата поведения включаемого варианта использования или значения, получаемого после его выполнения.

В некотором смысле выполнение включаемого варианта использования аналогично вызову подпрограммы. При этом поведение включаемого варианта использования должно быть закончено, прежде чем будет завершено выполнение базисного варианта использования или вызвано выполнение другого включаемого варианта использования.

На практике отношение включения применяется для моделирования ситуаций, когда существуют общие части поведения двух или более вариантов использования. Тогда эта общая часть может быть выделена в отдельный вариант использования, чтобы затем включить его во все базисные варианты использования, имеющие эту часть в своем поведении. При этом главное назначение отношения включения связано с повторным использованием этой общей части, которая не указывается в базисных вариантах использования. В этом случае функциональность базисных вариантов использования не является полной сама по себе, но зависит от включаемых частей, имеющих самостоятельное поведение. Это отражается в направлении данного отношения, которое указывает, что базисный вариант использования зависит от включаемого варианта использования, но не наоборот.

В общем случае один базовый вариант использования может быть связан отношением включения с несколькими включаемыми вариантами использования, а также самому быть включаемым в другие варианты использования. При этом включаемый вариант использования предоставляет базовому варианту использования некоторое инкапсулированное поведение, детали которого могут быть скрыты от базового варианта использования. Это инкапсули-

рованное поведение, в свою очередь, может быть также распределено между несколькими включаемыми вариантами использования.

Разработчикам следует помнить, что на одной диаграмме вариантов использования не может быть замкнутого пути по отношению включения. В противном случае такая диаграмма будет являться семантически противоречивой или не согласованной (ill-formed). Практически все известные автору CASE-средства не позволяют выявить подобные и другие ошибки, о чем также следует знать и всем читателям.

3.3.3. Отношение расширения

Отношение расширения в языке UML 2.0 также является частным случаем общего отношения зависимости между двумя вариантами использования.

Отношение *расширения (extend)* определяет взаимосвязь одного варианта использования с некоторым другим вариантом использования, функциональность или поведение которого задействуется первым не всегда, а только при выполнении некоторых дополнительных условий.

Отношение расширения является направленным бинарным отношением, поскольку два варианта использования в этом отношении всегда являются упорядоченными.

Графически данное отношение обозначается как отношение зависимости в форме пунктирной линии с "V"-образной стрелкой, направленной от зависимого варианта использования к независимому варианту использования и соединенной с ним в так называемой *точке расширения* (extension point). При этом зависимый вариант использования в отношении расширения называется также *расширяющим*, а независимый — *базовым* или *расширяемым* вариантом использования. Линия со стрелкой обязательно помечается ключевым словом `<<extend>>`, которое записывается в форме стереотипа (рис. 3.9).

В изображенном фрагменте диаграммы (см. рис. 3.9), которая относится к модели интернет-магазина, представлено отношение расширения между базовым вариантом использования "Оформление Заказа в интернет-магазине" и расширяющим вариантом использования "Предоставление бонусной скидки постоянному покупателю". Наличие данного отношения расширения означает, что поведение или функциональность первого варианта использования

в некоторых случаях могут быть дополнены поведением или функциональностью второго варианта использования. Чтобы это расширение функциональности имело место, должно быть выполнено специальное логическое условие данного отношения расширения — например, наличие у покупателя бонусной карточки.

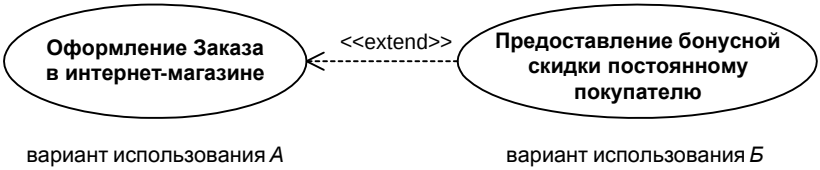


Рис. 3.9. Пример графического изображения отношения расширения между вариантами использования

В общем случае отношение расширения позволяет моделировать тот факт, что базовый вариант использования *А* может присоединять к своему поведению некоторое дополнительное поведение, определенное в форме расширяющего варианта использования *Б*. Наличие данного отношения всегда предполагает проверку некоторого условия и ссылку на точку расширения в базовом варианте использования. Точка расширения определяет место в базовом варианте использования, в которое должно быть вставлено поведение расширяющего варианта использования при выполнении соответствующего логического условия.

Следует заметить, что расширяемый вариант использования определяется и имеет содержательный смысл независимо от расширяющего варианта использования. С другой стороны, расширяющий вариант использования обычно определяет поведение, которое само по себе без базового варианта использования не имеет содержательного смысла. Иногда удобно считать, что расширяющий вариант использования определяет множество приращений модульного поведения, которые добавляются в выполнение расширяемого варианта использования при выполнении специальных условий. При этом само отношение расширения является собственностью расширяющего варианта использования.

Более подробно семантика отношения расширения определяется следующим образом. При выполнении последовательности действий, которая определяет поведение базового варианта использования, и наличии в нем некоторой точки расширения, которая является первой из всех точек расширения у базового варианта использования, проверяется логическое условие для данной точки расширения. Если это условие выполняется, исходная последовательность

действий базового варианта использования приостанавливается и начинается выполнение последовательности действий, определенных для расширяющего варианта использования.

Если же логическое условие первой точки расширения не выполняется, исходная последовательность действий базового варианта использования не приостанавливается и продолжает выполняться либо до своего окончания, либо до следующей точки расширения. При наличии других точек расширения у базового варианта использования выполняется последовательная проверка соответствующих условий с аналогичными последствиями, а в случае их отсутствия — базовый вариант использования выполняется до своего завершения, без какого бы то ни было прерывания.

Следует заметить, что логическое условие в каждой точке расширения проверяется лишь один раз — в момент первой ссылки на соответствующую точку расширения при выполнении последовательности действий базового варианта использования.

В общем случае точка расширения идентифицирует точку в поведении базового варианта использования, в которой его поведение может быть расширено поведением расширяющего варианта использования, как это определено в отношении расширения. Формально точка расширения может иметь некоторое имя, которое должно быть уникальным в базовом варианте использования. При этом точка расширения является ссылкой на местоположение в варианте использования, в которое должно быть вставлено поведение расширяющего варианта использования. Само объяснение, которое является необязательным, может быть любым неформальным текстом или более точной спецификацией точки расширения в базовом варианте использования в форме структурированного текста.

В представленном далее примере изображено отношение расширения между двумя вариантами использования, дополненное комментариями в форме примечаний. При этом для уточнения условия в соответствующей точке расширения служит пояснительный текст в произвольной форме (рис. 3.10).

В контексте рассматриваемой модели это означает, что при отсутствии у покупателя бонусной карточки поведение расширяющего варианта использования не задействуется и, соответственно, покупателю не предоставляется бонусная скидка. Поскольку в рассматриваемом фрагменте (см. рис. 3.10) имеется единственная точка расширения, то спецификация логического условия для точки расширения в форме примечания относится к базовому варианту использования. Примечание для включающего варианта использования в этом случае не является обязательным и служит для удобства чтения и интерпретации данного фрагмента диаграммы.

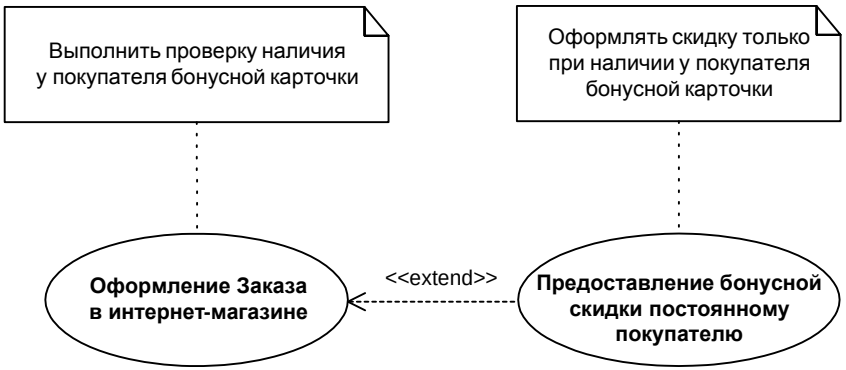


Рис. 3.10. Графическое изображение отношения расширения с условием выполнения в форме произвольного текста

Более строго точку расширения можно специфицировать следующим образом. К соответствующему отношению расширения присоединяется примечание, которое содержит запись условия в форме ограничения, заключенного в фигурные скобки. Ниже строки с условием после ключевого слова `extension point` и двоеточия записывается имя точки расширения. Имя данной точки расширения также должно быть указано и в базовом варианте использования после ключевого слова `extension point` (рис. 3.11).

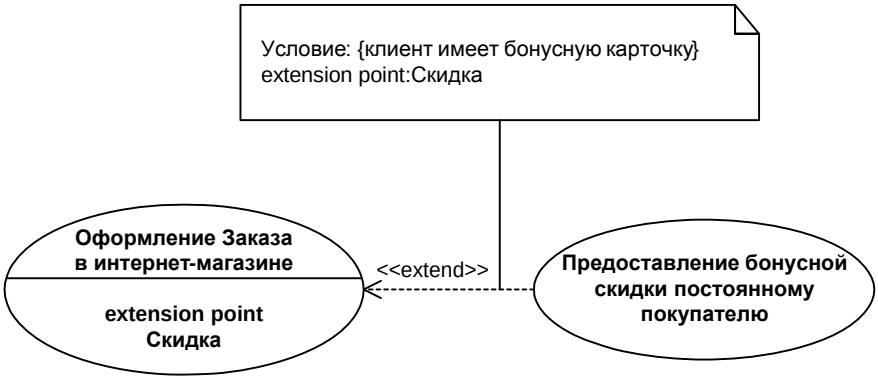


Рис. 3.11. Графическое изображение отношения расширения с условием выполнения в форме структурированного текста

Данный фрагмент диаграммы вариантов использования может быть интерпретирован следующим образом. Вариант использования "Оформление Заказа в интернет-магазине" имеет точку расширения "Скидка". Данный вариант

использования расширяется в этой точке расширения вариантом использования "Предоставление бонусной скидки постоянному покупателю" всякий раз, когда выполнение базового варианта использования доходит до места, на которое ссылается точка расширения "Скидка" и клиент имеет бонусную карточку. При этом вариант использования "Оформление Заказа в интернет-магазине" определяется независимо от варианта использования "Предоставление бонусной скидки постоянному покупателю".

В общем случае базовый вариант использования может иметь несколько точек расширения, с каждой из которых должен быть связан расширяющий вариант использования. Один расширяющий вариант использования, в свою очередь, может быть связан отношением расширения с несколькими базовыми вариантами использования, а также иметь в качестве собственных расширений другие варианты использования. Во всех подобных случаях разработчикам следует помнить, что поведение любого базового варианта использования не должно зависеть от поведения его расширений, как это следует из общей семантики данного отношения.

Разработчикам следует помнить, что на одной диаграмме вариантов использования не может быть замкнутого пути по отношению расширения. В противном случае такая диаграмма будет являться семантически противоречивой или несогласованной (ill-formed), что может служить признаком низкой квалификации разработчика в знании языка UML 2.0.

3.3.4. Отношение обобщения

Отношение *обобщения (generalization)* предназначено для спецификации того факта, что один элемент модели является специальным или частным случаем другого элемента модели.

Главной особенностью отношения обобщения является то, что оно может связывать между собой только элементы одного типа. Графически отношение обобщения обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на общий элемент модели.

В изображенном на рис. 3.12 фрагменте диаграммы вариантов использования отношение обобщения специфицирует то, что вариант использования "Оплата товара по кредитной карточке" является специальным случаем варианта использования "Оплата выбранного в интернет-магазине товара". При этом вариант использования *Б*, который является потомком, наследует все свойства поведения своего родителя, т. е. варианта использования *А*.

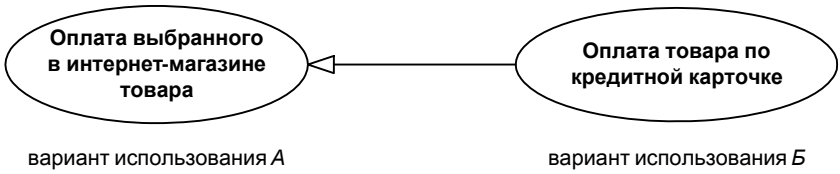


Рис. 3.12. Пример графического изображения отношения обобщения между вариантами использования

Применительно к вариантам использования семантика данного отношения указывает на тот факт, что вариант использования *Б* является специализацией варианта использования *А*. При этом вариант использования *А* называется *предком* или *родителем* по отношению к варианту использования *Б*, в свою очередь *Б* называется *потомком* или *дочерним* по отношению к варианту использования *А*. Следует подчеркнуть, что потомок наследует все свойства поведения своего родителя, а также может обладать дополнительными особенностями поведения.

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов. При этом потомки участвуют во всех отношениях родительских вариантов использования. В свою очередь, потомки могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения.

В языке UML 2.0 один вариант использования может иметь несколько родительских вариантов использования. В этом случае реализуется так называемое множественное наследование свойств и поведения всех родительских вариантов использования. С другой стороны, один вариант использования может быть предком для нескольких дочерних вариантов использования, что соответствует таксономическому характеру отношения обобщения.

Между актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера *Б* к актеру *А* отмечает тот факт, что каждый экземпляр актера *Б* является одновременно экземпляром актера *А* и обладает всеми его свойствами. В этом случае актер *А* является родителем по отношению к актеру *Б*, а актер *Б*, соответственно, потомком актера *А*. При этом актер *Б* обладает способностью играть такое же множество ролей, что и актер *А*. Графически данное отношение также обозначается стрелкой обобщения, которая указывает на родительского актера (рис. 3.13).

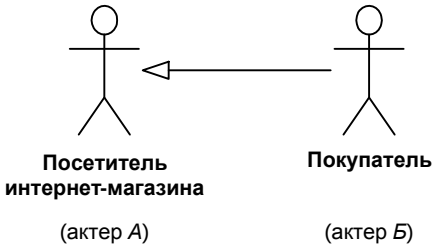


Рис. 3.13. Пример графического изображения отношения обобщения между актерами

В данном фрагменте диаграммы вариантов использования отношение обобщения служит для указания того факта, что актер "Покупатель" является специальным случаем актера "Посетитель интернет-магазина". При этом актер *Б*, который является потомком, наследует все свойства своего родителя, т. е. актера *А*.

На практике два и более актера могут иметь общие свойства, например, они могут взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения может быть представлена в форме отношения обобщения с другим, возможно, абстрактным актером, который моделирует соответствующую общность ролей.

3.3.5. Пример диаграммы вариантов использования для системы продажи товаров в интернет-магазине

Для иллюстрации применения рассмотренных элементов графической нотации языка UML 2.0 построим диаграмму вариантов использования для системы продажи товаров в интернет-магазине. Процесс функционирования интернет-магазина достаточно хорошо знаком всем пользователям Интернета, поэтому не требует пространного описания. Эта модель может быть использована при разработке и реализации интернет-проектов.

В качестве основного актера описываемой системы можно рассматривать посетителя интернет-магазина. Посетитель интернет-магазина может просматривать список товаров интернет-магазина, помещать товар в виртуальную корзину и изменять содержимое этой корзины. Посетитель может стать покупателем, если он принимает решение об оформлении заказа на покупку выбранных им товаров. Как следует из существа выдвигаемых к системе

требований, этот сервис выступает в виде основного, поэтому в качестве базового варианта использования данной модели может служить вариант использования "Оформление Заказа на покупку товаров".

В качестве других актеров рассматриваемой системы могут выступать менеджер интернет-магазина и бухгалтер. При этом менеджер может изменять список товаров и специфицировать условия для предоставления бонусной скидки, а бухгалтер — принимать оплату за выбранный покупателем товар.

Поскольку при оформлении заказа на покупку товара необходима регистрация покупателя, и эта функциональность выполняется всегда, она может быть выделена в отдельный вариант использования, который будет связан с базовым отношением включения. С другой стороны, при оформлении заказа постоянному покупателю может быть предоставлена специальная бонусная скидка. Это требование может быть также представлено в качестве отдельного варианта использования "Предоставление бонусной скидки", который будет связан с базовым отношением расширения.

Дальнейшая детализация модели может быть выполнена на основе установления дополнительного отношения обобщения для варианта использования "Оплата выбранного товара". Если в рамках рассматриваемой системы продажи товаров в интернет-магазине возможна оплата выбранных покупателем товаров, как наличными, так и по кредитной карточке, то в этом случае диаграмма может быть дополнена соответствующими вариантами использования. При этом варианты использования "Оплата товара наличными" и "Оплата товара по кредитной карточке" будут связаны с вариантом использования "Оплата выбранного товара" отношением обобщения.

Полученная в результате диаграмма вариантов использования будет содержать 9 вариантов использования и 4-х актеров, между которыми установлены соответствующие отношения включения, расширения и обобщения (рис. 3.14). Все варианты использования заключены в прямоугольник, который служит для обозначения субъекта проектируемой системы.

Поскольку диаграмма вариантов использования по своему назначению имеет концептуальный характер, на ней не могут быть изображены последовательности действий, которые необходимо выполнить при реализации соответствующих вариантов использования. Это обстоятельство служит основой для дополнения диаграммы вариантов использования специальными текстовыми сценариями, которые предназначены для неформального описания подобной последовательности действий. Рекомендации и шаблоны написания текстовых сценариев рассматриваются далее в этой главе.

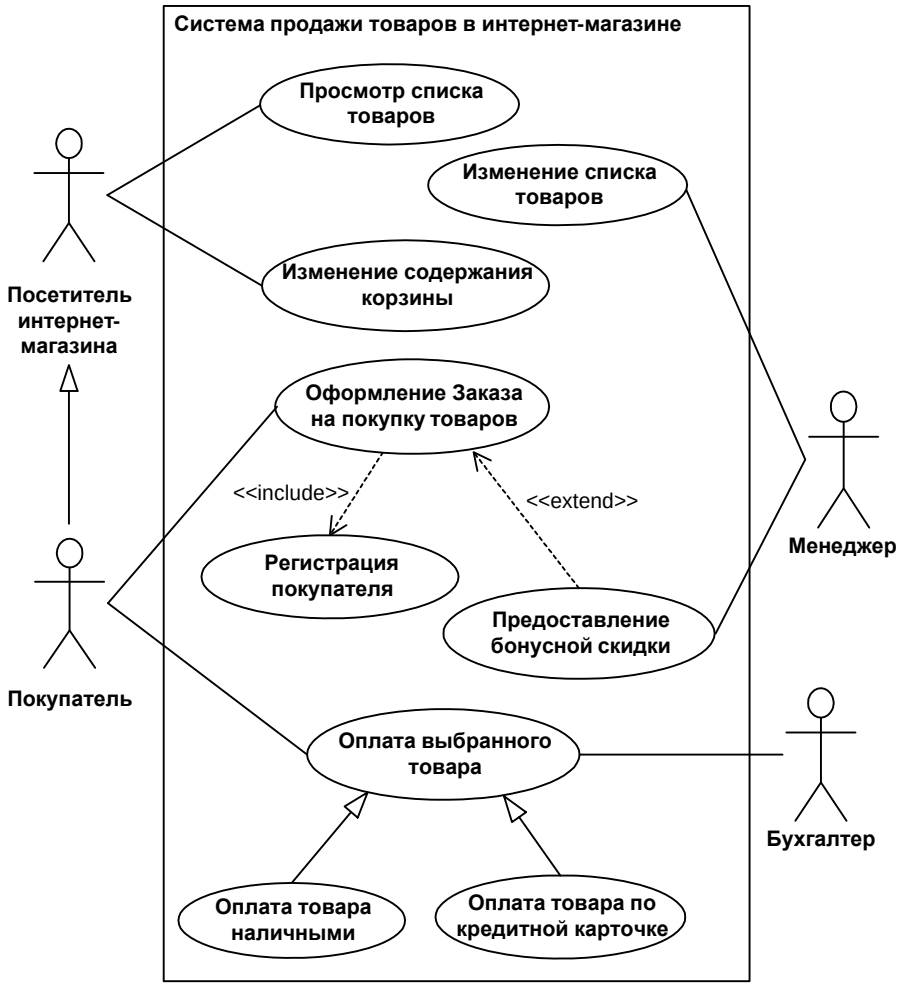


Рис. 3.14. Диаграмма вариантов использования для системы продажи товаров в интернет-магазине

Следует заметить, что построенная диаграмма вариантов использования не содержит функциональности, связанной с доставкой товара покупателю и поддержанием нужного запаса товаров на складе интернет-магазина, а также спецификации точки расширения. Дополнить представленную модель соответствующими вариантами использования, актерами и точкой расширения предлагается читателям самостоятельно в качестве упражнения.

3.4. Формализация функциональных требований к системе с помощью диаграммы вариантов использования

Варианты использования являются средством для спецификации требований к системе, т. е. того, что проектируемая система предположительно должна делать. Требуемое поведение системы или субъекта специфицируется одним или несколькими вариантами использования, которые определяются в соответствии с потребностями актеров. Эти поведения, включая взаимодействия между актерами и субъектом, могут привести в результате к изменению состояния субъекта и коммуникации с его окружением.

Требование (requirement) — желательное свойство, характеристика или условие, которым должна удовлетворять система в процессе своей эксплуатации.

На практике при разработке программных систем рекомендуется придерживаться следующего правила — отдельному варианту использования должно соответствовать некоторое требование к функциональному поведению моделируемой системы.

3.4.1. Классификация требований в модели FURPS+

Применительно к программным системам предложена следующая классификация требований, которая получила название модели FURPS+, что соответствует первым буквам соответствующих категорий требований на английском языке:

- функциональные требования (Functionality);
- требования удобства использования (Usability);
- требования надежности (Reliability);
- требования производительности (Performance);
- требования удобства сопровождения (Supportability).

При этом символом "+" обозначены дополнительные требования, к которым относятся:

- проектные ограничения;
- требования управления системой;

- требования к графическому интерфейсу пользователя;
- физические требования;
- юридические требования.

Центральное место среди указанных требований занимают функциональные требования, которые в контексте моделей UML 2.0 должны служить исходной информацией для построения диаграмм вариантов использования. Однако на практике одной графической нотации языка UML 2.0 может оказаться недостаточно для спецификации функциональных требований.

Построение диаграммы вариантов использования является самым первым этапом процесса объектно-ориентированного анализа и проектирования, цель которого — представить совокупность функциональных требований к поведению проектируемой системы. Спецификация требований к проектируемой системе в форме диаграммы вариантов использования и дополнительных сценариев может представлять собой самостоятельную модель, которая в языке UML получила название *модели вариантов использования* и имеет свое специальное стандартное имя или стереотип `<<useCaseModel>>`.

В последующем все заданные в этой модели требования могут быть представлены в виде общей модели системы, которая может быть оформлена в виде отдельного пакета с именем `Система`. Этот пакет в свою очередь может представлять собой иерархию пакетов, на самом верхнем уровне которой содержится множество классов, реализующих базовые варианты использования проектируемой системы. При этом пакет системы самого верхнего уровня может быть дополнительно помечен стереотипом `<<topLevelPackage>>`.

Одним из требований самого языка UML 2.0 является самодостаточность графических диаграмм для представления информации о моделях проектируемых систем. Однако большинство разработчиков и экспертов согласны с тем, что изобразительных средств языка UML 2.0 недостаточно, чтобы учесть на диаграммах вариантов использования особенности функционального поведения достаточно сложной системы. С этой целью рекомендуется дополнять этот тип диаграмм текстовыми сценариями, которые уточняют или детализируют последовательность действий, совершаемых системой при реализации своего поведения.

3.4.2. Спецификация функциональных требований с помощью текстовых сценариев

Содержание варианта использования может быть дополнительно специфицировано в форме пояснительного текста, который раскрывает смысл или семантику выполняемых действий при выполнении варианта использования.

Такой пояснительный текст применительно к диаграммам вариантов использования получил специальное название текста-сценария или просто — сценария.

Сценарий (scenario) — специально написанный текст, который описывает поведение моделируемой системы в форме последовательности выполняемых действий актеров и самой системы.

В настоящее время разработчикам предлагаются различные способы или шаблоны для написания подобных сценариев. Один из таких шаблонов рассматривается далее и может быть рекомендован читателям для применения на начальных этапах концептуального моделирования (табл. 3.1).

Таблица 3.1. *Шаблон для написания сценария отдельного варианта использования*

Главный раздел	**Раздел "Типичный ход событий"**	**Раздел "Исключения"**	**Раздел "Примечания"**
Имя варианта использования Актеры Цель Краткое описание Тип Ссылки на другие варианты использования	Типичный ход событий, приводящий к успешному выполнению данного варианта использования	Исключение № 1	Примечание № 1
		Исключение № 2	Примечание № 2
		...	...
		Исключение № *n*	Примечание № *n*

Для удобства записи и чтения сценария часто используют модификацию данного шаблона, располагая разделы последовательно сверху вниз. При этом написание сценариев модели начинают с базовых или основных вариантов использования, после чего рассматриваются сценарии второстепенных или включаемых вариантов использования. Сценарии расширяющих вариантов использования обычно помещаются в разделе исключений базового варианта использования. Желательно, чтобы каждому исключению соответствовал отдельный вариант использования, который соединяется с базовым отношением расширения. В противном случае будет нарушено логическое соответствие между текстами сценариев и диаграммой вариантов использования.

Если поведение некоторого варианта использования представляется разработчику тривиальным, то и написание для него сценария не является необхо-

димым. В этом случае можно ограничиться простым примечанием к соответствующему исключению. При написании сценариев вариантов использования важно понимать, что сценарий должен дополнять или уточнять диаграмму вариантов использования, но не заменять ее полностью. В противном случае будут потеряны достоинства визуального представления моделей.

Некоторые разработчики, с оптимизмом воспринявшие работу А. Кокберна по написанию эффективных вариантов использования, рекомендуют вообще отказаться от графического изображения последних. Судя по всему, адепты текстовых сценариев забывают о существовании в языке UML 2.0 других типов диаграмм, поскольку их основным аргументом против изображения диаграмм вариантов использования является перегруженность или избыток визуальных элементов. Действительно, при разработке диаграмм вариантов использования следует вовремя остановиться и не пытаться на этапе концептуального моделирования изображать логику или последовательность выполнения отдельных действий. Для последней цели в языке UML 2.0 специально предназначены диаграммы последовательности и деятельности, которые рассматриваются далее в книге.

3.4.3. Пример сценария для системы продажи товаров в интернет-магазине

Для иллюстрации рассмотренного материала дополним разработанную ранее диаграмму вариантов использования для системы продажи товаров в интернет-магазине (см. рис. 3.14) текстовым сценарием, написанным на основе предложенного шаблона (см. табл. 3.1). Сценарий будет дополнять построенную диаграмму, раскрывая содержание отдельных действий и их логическую последовательность, что характеризует внешнее представление о процессе функционирования системы.

В качестве примера выберем достаточно простой вариант использования "Просмотр списка товаров", для которого соответствующий сценарий удобно представить в виде двух отдельных таблиц, каждая из которых описывает отдельный раздел шаблона. В главном разделе сценария (табл. 3.2) указывается имя рассматриваемого варианта использования, имена взаимосвязанных с ним актеров, цель выполнения, тип и ссылки на другие варианты использования.

Таблица 3.2. *Главный раздел сценария выполнения варианта использования "Просмотр списка товаров"*

Вариант использования	**Просмотр списка товаров**
Актеры	Посетитель интернет-магазина
Краткое описание	Получение требуемой информации о товарах, представленных в интернет-магазине
Цель	Посетитель интернет-магазина просматривает информацию о товарах. Система обеспечивает доступ к любому товару и удобную навигацию по различным категориям товаров
Тип	Базовый
Ссылки на другие варианты использования	Отсутствуют

Поскольку для варианта использования "Просмотр списка товаров" отсутствуют включаемые варианты использования, ссылки на другие варианты использования также будут отсутствовать. В следующем разделе сценария (табл. 3.3) описывается последовательность действий, приводящая к успешному выполнению рассматриваемого варианта использования. В качестве инициатора действий выступает актер "Посетитель интернет-магазина". Для удобства каждое действие помечается порядковым номером в последовательности.

Таблица 3.3. *Раздел типичного хода событий сценария выполнения варианта использования "Просмотр списка товаров"*

Действия актеров	**Отклик системы**
1. Посетитель загружает исходную страницу интернет-магазина в браузер	2. Система отображает исходную страницу интернет-магазина
3. Посетитель интернет-магазина выбирает категорию интересуемых товаров	4. Система отображает информацию о выбранной категории товаров
5. Посетитель интернет-магазина выбирает интересуемый товар	6. Система отображает общую информацию о выбранном товаре
7. Посетитель интернет-магазина выбирает просмотр детальной информации об интересуемом товаре	8. Система отображает детальную информацию о выбранном товаре
9. Посетитель интернет-магазина может пожелать вернуться на исходную страницу интернет-магазина	10. Система отображает исходную страницу интернет-магазина

Поскольку для рассматриваемого варианта использования отсутствуют расширяющие варианты использования, секции исключений останутся пустыми. Заинтересованным читателям в качестве упражнения для самостоятельной работы предлагается написать аналогичные текстовые сценарии для остальных вариантов использования, представленных на ранее построенной диаграмме (см. рис. 3.13).

Общие рекомендации по разработке диаграмм вариантов использования, а также перечень типичных ошибок, которые встречаются при их разработке, хотя и не относятся непосредственно к семантике языка UML 2.0, могут оказать существенную помощь разработчикам. При необходимости с этим материалом можно познакомиться по ранее изданным книгам автора, перечень которых приводится в конце книги.

* * *

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут применяться при построении диаграмм вариантов использования (табл. 3.4).

Таблица 3.4. *Графические узлы и пути на диаграммах вариантов использования*

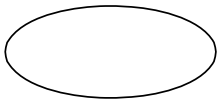
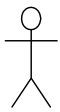
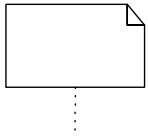
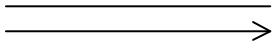
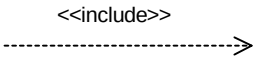
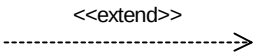
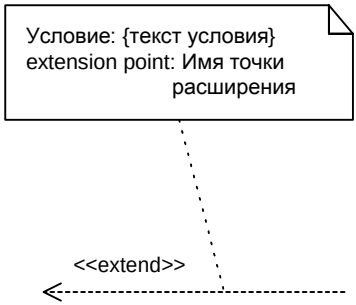
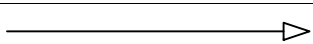
Тип графического элемента	Нотация	Ссылка на раздел
Вариант использования (use case)		Вариант использования *(см. разд. 3.2.1)*
Актер (actor) по умолчанию		Актер *(см. разд. 3.2.2)*
Актер (actor) — пример пиктограммы, определенной пользователем		Актер *(см. разд. 3.2.2)*
Субъект (subject)		Субъект вариантов использования *(см. разд. 3.1.2)*

Таблица 3.4 *(окончание)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Примечание (note)		Комментарий *(см. разд. 3.2.3)*
Ассоциация (association)		Отношение ассоциации *(см. разд. 3.3.1)*
Включение (include)		Отношение включения *(см. разд. 3.3.2)*
Расширение (extend)		Отношение расширения *(см. разд. 3.3.3)*
Расширение (с условием)		Отношение расширения *(см. разд. 3.3.3)*
Точка расширения (extension point)		Отношение расширения *(см. разд. 3.3.3)*
Обобщение (generalization)		Отношение обобщения *(см. разд. 3.3.4)*

Глава 4

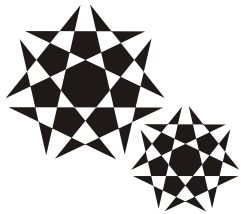


Диаграмма классов (class diagram)

Центральное место при проектировании программной системы занимает разработка ее логической модели в виде диаграммы классов. Графическая форма представления классов в языке UML 2.0 проста и интуитивно понятна всем тем, кто когда-либо имел опыт работы с CASE-средствами. Важность диаграммы классов основывается на том обстоятельстве, что данная диаграмма является основой для автоматической генерации программного кода во всех известных CASE-средствах, которые обладают подобной возможностью.

4.1. Диаграмма классов — основная логическая модель проектируемой системы

Диаграмма классов (class diagram) — диаграмма, предназначенная для представления модели статической структуры программной системы в терминологии классов объектно-ориентированного программирования.

Разработка диаграммы классов преследует следующие цели:

- определить сущности предметной области и представить их в форме классов с соответствующими атрибутами и операциями;
- определить взаимосвязи между сущностями предметной области и представить их в форме типовых отношений между классами;
- разработать исходную логическую модель программной системы для ее последующей реализации в форме физических моделей;
- подготовить документацию для последующей разработки программного кода.

В общем случае диаграмма классов представляет собой граф, вершинами или узлами которого являются элементы типа "классификатор", связанные различными типами структурных отношений.

Классификатор (classifier) — специальное понятие, предназначенное для классификации экземпляров, которые имеют общие характеристики.

Классификатор в языке UML 2.0 является абстрактным метаклассом. Классификатор является типом и может иметь собственные обобщения, тем самым делая возможным определение отношения обобщения с другими классификаторами. Он может специфицировать иерархию обобщения посредством ссылок на другие классификаторы. Классификатор является переопределяемым элементом, к которому может быть добавлена дополнительная семантика.

В общем случае классификатор определяет некоторый тип и является пространством имен, элементы которого могут иметь характеристики. Классификатор может быть абстрактным, который предназначен для использования другими классификаторами, например в качестве цели отношений обобщения.

Классификатор является абстрактным элементом модели и поэтому, строго говоря, не имеет нотации. Тем не менее для удобства в языке UML 2.0 для классификатора определена нотация, которая может быть использована для любого частного случая или специализации классификатора. Такой нотацией по умолчанию для классификатора является прямоугольник с непрерывным контуром, содержащий имя классификатора и необязательные секции, разделенные горизонтальными линиями, в которых содержатся характеристики или другие члены классификатора.

Имя классификатора записывается по центру полужирным шрифтом. Первая буква в имени классификатора должна быть заглавной, если множество используемых символов поддерживает верхний регистр. Имя абстрактного классификатора изображается курсивом. Абстрактный классификатор может быть также изображен с использованием ключевого слова `abstract`, указанным после или ниже имени классификатора. Атрибуты классификатора изображаются как строки текста. Специальный тип классификатора может быть указан в угловых кавычках выше его имени. Некоторые специализации классификатора имеют свои собственные графические нотации.

В общем случае классификатор может обладать некоторыми характеристиками, которые представляются обычным списком.

Характеристика (feature) — понятие, предназначенное для спецификации особенностей структуры и поведения экземпляров классификаторов.

Характеристика в языке UML 2.0 является абстрактным метаклассом. Характеристика может относиться к индивидуальным экземплярам классификатора (не статическая) или к самому классификатору (статическая). Одна и та же характеристика не может быть статической в одном контексте и не статической в другом. По умолчанию предполагается, что все характеристики являются не статическими.

Общая нотация для представления характеристики отсутствует. Различные подклассы определяют свою специальную нотацию. Подчеркиваются имена только статических характеристик. Многоточие "..." в качестве последнего элемента списка характеристик указывает на то, что существуют дополнительные характеристики, но они не указаны в этом списке.

В метамодели языка UML 2.0 различают два вида характеристик: структурные характеристики, которые применительно к классам называют свойствами или атрибутами, и характеристики поведения, которые применительно к классам называют операциями.

Структурная характеристика (structural feature) является типизированной характеристикой классификатора, которая специфицирует структуру его экземпляров.

Структурная характеристика предназначена для представления того факта, что экземпляры соответствующего классификатора имеют слот для записи и хранения одного или нескольких значений определенного типа. Формальный синтаксис для записи структурных характеристик отсутствует. Применительно к классам в языке UML 2.0 определена специальная нотация для записи атрибутов, которая рассматривается далее.

Структурная характеристика, не допускающая модификации или изменения своих значений, называется также *только для чтения* и изображается с использованием ключевого слова `readOnly` в качестве части нотации этой структурной характеристики. По умолчанию предполагается, что структурная характеристика допускает модификацию своих значений.

Характеристика поведения (behavioral feature) является характеристикой классификатора, которая специфицирует некоторый аспект поведения его экземпляров.

Характеристика поведения специфицирует то, как экземпляр классификатора будет реагировать на полученный запрос посредством вызова связанного с ней поведения. С этой целью характеристика поведения может содержать список параметров, описывающих порядок, тип и направление аргументов, которые могут быть заданы, когда данная характеристика поведения вызывается, или возвращаются, когда характеристика поведения заканчивается. Формальный синтаксис для записи характеристик поведения отсутствует. Применительно к классам в языке UML 2.0 определена специальная нотация для записи операций, которая рассматривается далее.

Диаграмма классов может содержать пакеты и интерфейсы, а также отношения между ними. Когда говорят о специфике данной диаграммы, то имеют в виду статическую модель структуры проектируемой системы. Поэтому диаграмму классов принято считать графическим представлением структурных взаимосвязей логической модели системы, которые не зависят от времени. Именно по этой причине на данной диаграмме не указывается информация о временных аспектах функционирования системы.

В общем случае статическая модель структуры достаточно сложной системы может быть представлена в виде нескольких диаграмм классов. С целью удобства диаграммы классов могут быть вложены в пакеты, которые соответствуют отдельным подсистемам. Это приводит к иерархическому представлению общей логической модели в форме вложенных диаграмм классов и пакетов. При этом каждая из диаграмм классов строится с учетом возможности ее повторного использования и удобства реализации на том или ином языке программирования.

Схожая нотация применяется для объектов, которые в контексте языка UML 2.0 являются экземплярами классов и изображаются на канонических диаграммах различных типов. Нотация UML 2.0 также предоставляет широкие возможности для отображения элементов с расширенной семантикой, таких как дополнительные стереотипы классов и отношений, предоставляемые и требуемые интерфейсы, абстрактные и параметризованные классы. Все эти элементы служат предметом рассмотрения настоящей главы.

4.2. Класс

Класс (class) — элемент модели, который описывает множество объектов, имеющих одинаковые спецификации характеристик, ограничений и семантики.

В языке UML 2.0 класс служит графическим представлением и формальным описанием свойств однотипных объектов, которые обладают одинаковой

структурой, поведением и отношениями с объектами из других классов. Цель класса заключается в спецификации классификации объектов и в спецификации характеристик, которые отражают структуру и поведение этих объектов.

Для графического изображения класса в языке UML 2.0 используется символ классификатора — прямоугольник, при этом ключевое слово `class` в угловых скобках выше его имени не указывается. При этом символ классификатора без ключевого слова `metaclass` в угловых кавычках указывает на то, что это класс.

Прямоугольник классификатора для класса дополнительно может быть разделен горизонтальными линиями на разделы или секции. В этих секциях указываются имя класса, атрибуты (структурные характеристики или свойства), операции (характеристики поведения), а также дополнительная информация, такая как ограничения или помеченные значения. Любая секция может быть опущена или скрыта. При этом разделительная линия для опущенной секции не изображается. Если некоторая секция опущена или скрыта, нельзя ничего сказать о наличии или отсутствии элементов в этой секции. Если необходимо, то для устранения двусмысленности могут быть использованы имена секций. Тип, видимость, значение по умолчанию, кратность, строка-свойство могут не отображаться, даже если они существуют в модели.

Рис. 4.1. Варианты графического изображения класса на диаграмме классов: *а* — в виде прямоугольника; *б* — в виде прямоугольника с секциями; *в* — без секции атрибутов класса; *г* — активный класс

Обязательным элементом при обозначении класса является его имя, которое всегда записывается в верхней секции прямоугольника класса. При этом класс может обозначаться прямоугольником без дополнительных секций с указанием только имени класса (рис. 4.1, *а*), что характерно для начальных этапов разработки диаграммы. По мере дальнейшей разработки диаграммы классов их обозначения дополняются секциями атрибутов и операций (рис. 4.1, *б*). Если секция атрибутов или операций является пустой, то в языке UML 2.0 допускается ее вовсе не указывать (рис. 4.1, *в*).

Дополнительно класс может быть специфицирован как активный или пассивный.

Активный класс (active class) — класс, каждый экземпляр которого имеет свою собственную нить управления.

Пассивный класс (passive class) — класс, каждый экземпляр которого выполняется в контексте некоторого другого объекта.

Объект активного класса после своего создания начинает исполнять поведение своего класса и не прекратит его до тех пор, пока либо не будет закончено выполнение этого поведения, либо данный объект будет уничтожен некоторым другим объектом. Активный класс в языке UML 2.0 изображается прямоугольником класса с дополнительным вертикальным отрезком на каждой его стороне (рис. 4.1, *г*).

Примеры графического изображения конкретных классов приведены на рис. 4.2. В первом случае для класса `Линия` (рис. 4.2, *а*) указаны только его атрибуты — координаты двух точек в некоторой системе координат, которые определяют расположение линии. Для класса `Окно` (рис. 4.2, *б*) указаны только операции, при этом секция его атрибутов не изображена. Для класса `Счет` (рис. 4.2, *в*) изображена только четвертая секция, в которой указано исключение — неверная аутентификация клиента. При этом секции атрибутов и операций не изображены.

Рис. 4.2. Примеры графического изображения конкретных классов:
а — для класса `Линия` указаны атрибуты; *б* — для класса `Окно` указаны операции;
в — для класса `Счет` указано исключение

Спецификация отдельных элементов класса в языке UML 2.0 подчиняется достаточно строгим правилам, незнание которых зачастую приводит разработчиков к ошибкам. Далее в этом разделе подробно рассматриваются правила спецификации имен классов, атрибутов и операций.

4.2.1. Имя класса

Имя класса (class name) — строка текста, предназначенная для идентификации класса на диаграмме классов.

Имя класса должно быть уникальным в пределах пакета, который может содержать одну или несколько диаграмм классов. Имя класса указывается в самой верхней секции символа прямоугольника, поэтому эта секция часто называется секцией имени класса. Имя класса записывается в одну строку по центру полужирным шрифтом. Для языков, в которых различаются прописные и строчные буквы, имена классов записываются с прописной буквы. Имена из нескольких слов часто образуются с помощью конкатенации слов и с использованием строчных букв для всех слов, за исключением первых букв каждого слова, которые записываются прописными.

В секции имени класса могут записываться также стереотипы и помеченные значения. При этом текстовый стереотип класса всегда записывается выше строки имени класса по центру в угловых кавычках, а помеченное значение — ниже строки имени класса. В форме помеченных значений может представляться информация о разработчике данного класса, статус состояния разработки или номер версии. В качестве имен классов, как правило, используются существительные, записанные по практическим соображениям без пробелов. Однако более конкретные рекомендации относительно выбора имен классов основываются на рассмотрении типа выполняемого проекта.

В последнее время среди отечественных разработчиков принято все проекты, выполняемые с использованием нотации языка UML 1.*x*/2.0, разделять на два типа: проекты без генерации программного кода и проекты с генерацией программного кода. Для проектов первого типа диаграмма классов служит моделью анализа. Имена классов в этом случае можно записывать символами кириллицы, т. е. использовать русскоязычные слова, возможно, с пробелами. Это значительно повышает наглядность диаграммы классов для всех заинтересованных лиц.

Для проектов второго типа диаграмма классов служит моделью проектирования. Имена классов в этом случае следует записывать символами латиницы, т. е. использовать англоязычные слова без пробелов. При этом имена классов должны соответствовать правилам синтаксиса выбранного языка генерации программного кода.

Хотя согласно требованиям спецификации языка UML 2.0 имя класса должно записываться полужирным шрифтом, это условие соблюдается разработчиками крайне редко. Возможно, причиной служит тот факт, что многие CASE-средства не поддерживают данного правила. В дальнейшем толщине шрифта при записи имени класса не будет придаваться принципиального значения.

Примерами имен классов могут быть существительные предметной области, такие как `Компания`, `Сотрудник`, `Проект`, `Заказ`, имеющие непосредственное отношение к моделируемой предметной области. Примерами имен классов также могут быть имена классов используемых библиотек, например, `CObject`, `CRecordView`, `TObject`, `TMenu`, соответствующих функциональному назначению отдельных элементов проектируемой системы.

Если предполагается программная реализация модели проектируемой системы, то очевидно имена классов в этом случае следует записывать на английском, при этом возможно учитывать и синтаксические особенности языка реализации. В рассматриваемых в книге примерах имена классов записываются на русском из методических соображений. Использование CASE-средств для автоматизации ООАП, чаще всего, накладывает свои собственные требования на язык спецификации моделей.

Класс может не иметь экземпляров или объектов. В этом случае он называется *абстрактным* (abstract) классом, а для обозначения его имени используется наклонный шрифт (курсив). В языке UML 2.0 принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом.

Курсив при записи имени класса, в отличие от толщины шрифта, имеет принципиальное значение, поскольку является семантическим аспектом описания абстрактных элементов языка UML 2.0. Именно по этой причине разработчикам следует внимательно записывать имена классов.

В некоторых случаях необходимо явно указать, к какому пакету относится тот или иной класс. Для этой цели используется так называемое *квалифицированное имя* (qualified name) класса со специальным символом в качестве разделителя имени — двойное двоеточие `::`. При этом имя класса без символа разделителя называется *простым именем* класса.

Синтаксис строки квалифицированного имени класса в общем случае имеет следующий вид:

`<`*`Имя пакета 1`*`>::<`*`Имя пакета 2`*`>::...::<`*`Имя пакета N`*`>::<`*`Простое имя класса`*`>`

где *`Имя пакета 1`*, *`Имя пакета 2`*, *`Имя пакета N`* — имена вложенных пакетов. Здесь *`Имя пакета N`* — имя пакета, которому непосредственно принадлежит специфицируемый класс. Другими словами, в квалифицированном имени класса перед простым именем должны быть явно указаны имена пакетов, в которых он определен.

Например, если класс `Счет` определен в пакете с именем `Счета Физических Лиц`, то квалифицированное имя этого класса может быть записано в виде: `Счета Физических Лиц::Счет`. Если же пакет `Счета Физических Лиц` в свою очередь вложен в пакет `Банк`, то квалифицированное имя рассматриваемого класса может быть записано в виде: `Банк::Счета Физических Лиц::Счет`.

4.2.2. Атрибуты класса

Атрибут (attribute) класса служит для представления отдельной структурной характеристики или свойства, которое является общим для всех объектов данного класса.

В общем случае атрибут представляет собой структурную характеристику или свойство, относящееся к классификатору или концу ассоциации. Атрибут устанавливает отношение экземпляра класса со значением или коллекций значений типа атрибута. Если свойство является атрибутом классификатора, то его значение или значения относятся к экземпляру классификатора посредством заполнения слотов этого экземпляра. Если свойство является собственностью классификатора, отличным от ассоциации, то оно представляет атрибут класса или типа данных.

Атрибуты класса записываются во второй сверху секции прямоугольника класса, которую называют *секцией атрибутов*. При этом все спецификации атрибутов выравниваются слева. Следует показывать полные спецификации атрибутов, когда это необходимо, и скрывать их в других случаях. Каждому атрибуту класса в языке UML 2.0 соответствует отдельная строка текста, которая в нотации БНФ имеет следующий формат:

```
<атрибут>::= [<видимость>] ['/'] <имя> [':' <тип атрибута>]
['['<кратность>']']  ['=' <значение по умолчанию>]
 ['{'<модификатор атрибута> [',' <модификатор атрибута>]* '}']
```

Здесь:

- `<видимость>::= '+' | '-' | '#' | '~'`. Другими словами, *видимость* (visibility) может принимать одно из 4-х возможных значений и отображаться либо посредством специального символа, либо соответствующего ключевого слова. Более подробно семантика значений видимости рассматривается далее;
- `'/'` означает, что атрибут является *производным* (derive). Значение производного атрибута может быть вычислено на основе значений других атрибутов этого или других классов. Поэтому данный атрибут называют иногда *вычислимым*. При использовании производных атрибутов разработчик должен явно указать процедуру или операцию для вычисления их значений;

- <*имя*> (name) представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом в обозначении атрибута, должно начинаться со строчной (малой) буквы и, как правило, не должно содержать пробелов. Относительно выбора языка для записи имени атрибута имеют место рекомендации, аналогичные рассмотренным выше рекомендациям по записи имен классов;
- <*тип атрибута*> (attribute type) есть имя классификатора, который является типом данного атрибута. Тип атрибута представляет собой имя некоторого типа данных, определенного или в пакете `Типы данных` языка UML 2.0, или самим разработчиком. Типы атрибутов классов часто соответствуют основным типам данных того языка программирования, который предполагается использовать для реализации данной модели. В общем случае тип атрибута записывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которому относится рассматриваемый класс. Типу атрибута должно предшествовать двоеточие;
- <*кратность*> (multiplicity) атрибута характеризует общее количество конкретных значений для атрибута, которые могут быть заданы для объектов данного класса. Более подробно семантика кратности рассматривается далее;
- <*значение по умолчанию*> (default) — некоторое выражение, которое служит для задания начального значения или значений данного атрибута в момент создания отдельного экземпляра соответствующего класса. Конкретное значение по умолчанию должно соответствовать типу данного атрибута. Если этот терм не указан, то значение атрибута на момент создания нового экземпляра класса не определено. С другой стороны, конструктор соответствующего объекта может переопределять исходное значение в процессе выполнения программы, если в этом возникает необходимость;
- <*модификатор атрибута*> (attribute modifier) представляет собой текстовое выражение, которое придает дополнительную семантику данному атрибуту. При этом набор возможных модификаторов атрибутов в языке UML 2.0 фиксирован и может быть представлен в следующем виде (БНФ):

```
<модификатор атрибута>::= 'readOnly' | 'union' | 'subsets'
  <имя атрибута>  | 'redefines' <имя атрибута> | 'ordered' |
  'unique' | <ограничение атрибута>
```

Назначение указанных модификаторов описываются в табл. 4.1.

Таблица 4.1. *Назначение модификаторов атрибутов в языке UML 2.0*

Модификатор атрибута	Назначение модификатора
`readOnly`	Атрибут является атрибутом только для чтения
`union`	Атрибут является производным объединением его подмножеств
`subsets` <*имя атрибута*>	Атрибут является собственным подмножеством атрибута с именем <*имя атрибута*>
`redefines` <*имя атрибута*>	Атрибут переопределяет некоторый наследуемый атрибут с именем <*имя атрибута*>
`ordered`	Значения атрибута являются упорядоченными. Этот порядок означает, что существует отображение из множества положительных целых чисел в элементы этой коллекции значений. Если атрибут не является многозначным, то значение для `ordered` не имеет семантического эффекта. При отсутствии этого модификатора атрибут специфицируется как неупорядоченный, и никаких предположений нельзя сделать относительно порядка его возможных значений
`unique`	Значения многозначного атрибута не могут иметь дубликатов, т. е. повторяться. Предполагается, что кратность соответствующего атрибута должна быть больше 1. Если атрибут не является многозначным, то значение `unique` не имеет семантического эффекта
<*ограничение атрибута*>	Выражение, которое специфицирует некоторое ограничение, применяемое к данному атрибуту

При задании атрибутов может быть использована дополнительная синтаксическая конструкция — подчеркивание строки атрибута. *Подчеркивание* строки атрибута означает, что значение соответствующего атрибута является статическим, т. е. одинаковым для всех экземпляров данного класса (аналог ключевого слова `static` в некоторых языках программирования).

Атрибуты или операции могут быть представлены сгруппированными в своих секциях по видимости. Ключевое слово или символ видимости в этом случае может быть указан один раз для нескольких характеристик с той же видимостью.

Вид видимости

Вид видимости является типом перечисления, который определяет значения в форме литералов для определения видимости элементов в модели. Вид видимости является перечислением следующих значений литералов:

- `public` (общедоступный). Общедоступный элемент является видимым всеми элементами, который имеют доступ к содержимому пространства имен, который им владеет;

- `private` (закрытый). Закрытый элемент является видимым только внутри пространства имен, который им владеет;
- `protected` (защищенный). Защищенный элемент является видимым для элементов, которые имеют отношение обобщения с пространством имен, который им владеет;
- `package` (пакет). Элемент, помеченный как имеющий пакетную видимость, является видимым всеми элементами в ближайшем охватывающем пакете. За пределами ближайшего охватывающего пакета элемент, помеченный как имеющий пакетную видимость, не является видимым.

Элемент, способный быть пакетом, является собственностью пространства имен, которое не является пакетом, и является видимым для элементов, которые расположены в этом же пакете, что и владеющее им пространство имен. Только именованные элементы, которые не находятся в собственности пакетов, могут быть помечены как имеющие пакетную видимость.

В случаях, когда именованный элемент может быть помечен несколькими видимостями, например импортируемый несколько раз, общедоступная видимость перекрывает закрытую видимость. Например, если элемент импортируется дважды в одно и то же пространство имен, один раз с использованием общедоступного импорта, а другой раз с помощью закрытого импорта, то он будет общедоступным.

Вместо литерала видимости в языке UML 2.0 допускается записывать соответствующий символ видимости:

- `+` — общедоступный;
- `-` — закрытый;
- `#` — защищенный;
- `~` — пакетный.

Видимость может быть не указана. Эта ситуация отличается от принятых по умолчанию соглашений в традиционных языках программирования, когда отсутствие квантора видимости трактуется как `public` или `private`.

Хотя язык UML 2.0 по определению инвариантен относительно реализации своих конструкций в конкретных языках программирования, на практике интерпретация значений видимости безотносительно к языкам программирования может привести к трудностям. Именно по этой причине значения видимости атрибутов и операций рекомендуется специфицировать в модели после того, как будет выбран язык ее программной реализации.

При реализации модели на конкретном языке программирования могут быть определены дополнительные значения видимости. Например, в языке C++ имеется видимость `friend` (дружественный), которая никак не специфицирована в языке UML 2.0. Подобные дополнения являются расширением базовой нотации и требуют от разработчика соответствующих пояснений в форме текста на естественном языке или в виде строки-свойства.

Кратность

Кратность (multiplicity) является спецификацией допустимой мощности множества при инстанцировании соответствующего элемента модели.

В языке UML 2.0 кратность может относиться к различным элементам модели (атрибутам, классам, концам ассоциаций). Кратность специфицируется как интервал целых чисел, начиная с нижней границы и заканчивая (возможно бесконечной) верхней границей. В общем случае спецификация кратности изображается в форме строки текста, содержащей границы интервала и необязательную спецификацию порядка и уникальности. Спецификация кратности в нотации БНФ имеет следующий формат:

```
<кратность> ::= <диапазон-кратности> [ '{' <указатель-упорядоченности>
['," <указатель-уникальности>] '}' ]
<диапазон-кратности> ::= [<нижняя-граница>'..' ] <верхняя-граница>
<нижняя-граница> ::= <целое число> | <спецификация значения>
<верхняя-граница> ::= '*' | <спецификация значения>
<указатель-упорядоченности> ::= 'ordered' | 'unordered'
<указатель-уникальности> ::= 'unique' | 'nonunique'
```

Здесь `<нижняя-граница>` должна быть неотрицательным целочисленным литералом, а `<верхняя-граница>` — произвольным (неограниченным) натуральным числом. При этом верхняя граница должна быть больше или равна нижней границе. Для представления неограниченной (бесконечной) верхней границы может быть использован символ "звездочка" `*` как часть спецификации кратности. Спецификация кратности для атрибута дополнительно помещается в квадратные скобки `[ ]`.

Общей нотацией для спецификаций упорядоченности и уникальности является использование строки-свойства, содержащей `ordered` или `unordered` для определения порядка, и `unique` или `nonunique` для определения уникальности. Для атрибутов соответствующие значения указываются в форме модификаторов (см. табл. 4.1).

Если нижняя граница равна верхней границе, то допускается использовать альтернативную нотацию в форме строки, содержащей только верхнюю границу. Например, `1` семантически эквивалентно `1..1`. Для кратности 0 в качестве

нижней границы и с не специфицированной верхней границей можно использовать эту альтернативную нотацию, указывая только знак звездочки * вместо записи 0..*. Кратность должна определять, по крайней мере, одну действительную мощность множества, т. е. быть больше 0.

Можно привести следующие примеры записи атрибутов для класса Сотрудник:

- + имяСотрудника : String {readOnly} — здесь имяСотрудника является именем общедоступного атрибута, который служит для представления информации только об имени конкретного сотрудника. Тип атрибута String (Строка) указывает на тот факт, что отдельное значение имени сотрудника представляет собой строку текста из одного слова (например, 'Кирилл' или 'Мария'). Модификатор readOnly указывает на тот факт, что после задания этого имени при инстанцировании объекта соответствующего класса оно не может быть изменено;
- ~ датаРождения : Data {readOnly} — атрибут с пакетной видимостью и типом Data (Дата), который должен быть дополнительно определен в модели. Модификатор readOnly указывает на тот факт, что после задания значения этого атрибута оно не может быть изменено;
- # /возрастСотрудника : Integer — производный атрибут с защищенной видимостью и простейшим типом Integer (Целое). Значение этого атрибута может быть вычислено как разность между текущей датой и датой рождения соответствующего сотрудника;
- + номерТелефона : Integer [*] {unique} — общедоступный атрибут, используемый в форме коллекции значений, когда у сотрудников рассматриваемой организации может быть несколько телефонных номеров, а может и не быть ни одного. На это указывает кратность атрибута в форме звездочки. В то же время среди номеров телефонов любого сотрудника не должно быть повторяющихся значений. Именно на этот факт указывает модификатор unique;
- - заработнаяПлата : Currency = $500 — закрытый атрибут с типом Currency (Денежный), который должен быть дополнительно определен в модели. При создании новых экземпляров класса Сотрудник, чему соответствует, например, прием сотрудника на работу, для него по умолчанию устанавливается заработная плата в $500. В последующем это значение может быть изменено.

4.2.3. Операции класса

Операция (operation) класса служит для представления отдельной характеристики поведения, которая является общей для всех объектов данного класса.

В общем случае операция является характеристикой поведения классификатора, которая специфицирует имя, тип, параметры и ограничения для вызова соответствующего поведения. Операция класса специфицирует некоторый сервис, который предоставляет каждый экземпляр или объект класса по требованию своих клиентов. В качестве клиентов операций могут выступать экземпляры различных классов, в том числе и экземпляры данного класса. Операции класса записываются в отдельной секции прямоугольника класса, которая, при наличии секции атрибутов, является третьей сверху. Эту секцию называют также *секцией операций*. Совокупность всех операций характеризует функциональный аспект поведения всех объектов данного класса.

Запись операций класса в языке UML 2.0 также стандартизована и подчиняется определенным синтаксическим правилам. Каждой операции класса соответствует строка текста, которую называют также *сигнатурой* операции. Общий формат записи отдельной операции класса следующий (БНФ):

```
<операция>::=[<видимость>] <имя операции> '(' [<список параметров>] ')'
 [':' [<тип возвращаемого результата>] '{' <свойство операции>
 [',' <свойство операции>]* '}'].
```

Далее назначение отдельных термов в записи операции рассматривается более подробно.

- `<видимость> ::= '+' | '-' | '#' | '~'`, как и в случае атрибута класса, может принимать одно из четырех возможных значений и изображаться либо в форме специального символа, либо соответствующего ключевого слова. Напомним семантику этих символов и соответствующих ключевых слов. Символ `+` обозначает операцию с видимостью "общедоступная" (`public`). Символ `#` обозначает операцию с видимостью "защищенная" (`protected`). Символ – используется для обозначения операции с видимостью "закрытая" (`private`). И, наконец, символ `~` применяется для обозначения операции с видимостью "пакетная" (`package`). Видимость для операции может быть опущена. Это означает, что видимость операции не определена.
- `<имя операции>` (operation name) представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной для каждой операции данного класса. Имя операции является единственным обязательным элементом в обозначении операции, должно начинаться со строчной (малой) буквы и, как правило, не должно содержать пробелов. После имени операции должны всегда следовать круглые скобки, даже если список параметров является пустым.

- <*список параметров*> (parameter list) представляет собой перечень разделенных запятыми формальных параметров операции и имеет следующий общий формат записи (БНФ):

  <*список параметров*>::= <*параметр*> [','<*параметр*>]*

  Более подробно спецификация параметра рассматривается далее.

- <*тип возвращаемого результата*> (return type) специфицирует тип значения, возвращаемого данной операцией. Двоеточие и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания нескольких возвращаемых значений данный элемент спецификации операции может быть записан в виде списка отдельных выражений.

- <*свойство операции*> (operation property) служит для указания дополнительных свойств, которые могут быть применены к данной операции. При этом набор возможных свойств операций в языке UML 2.0 фиксирован и может быть представлен в следующем виде (БНФ):

  <*свойство операции*>::= 'redefines' <*имя операции*> | 'query' | 'ordered' | 'unique' | <*ограничение*>

  Назначение свойств операций описываются в табл. 4.2.

Таблица 4.2. *Назначение свойств операций в языке UML 2.0*

Свойство операции	Назначение свойства операции
redefines <*имя операции*>	Данная операция переопределяет некоторую наследуемую операцию с именем <*имя операции*>
query	Данная операция не изменяет состояния моделируемой системы и, соответственно, не имеет побочного эффекта
ordered	Значения возвращаемого параметра являются упорядоченными. Предполагается, что кратность данного возвращаемого параметра должна быть больше 1
unique	Значения возвращаемого параметра не могут повторяться. Предполагается, что кратность данного возвращаемого параметра должна быть больше 1
<*ограничение*>	Выражение, которое специфицирует некоторое ограничение, применяемое к данной операции

Свойство операции не является обязательным, т. е. оно может отсутствовать, если никакие дополнительные свойства не специфицированы. Список параметров и тип возвращаемого значения также могут отсутствовать. В дополнение к рассмотренным правилам операция, область действия которой рас-

пространяется на все экземпляры класса, показывается подчеркиванием строки соответствующей операции.

Объявление или спецификация операции у некоторого класса на самом верхнем уровне абстракции позволяет наследовать данную операцию всеми потомками данного класса. Если же в некотором классе операция не реализуется, т. е. объекты данного класса не могут выполнять эту операцию, то такая операция называется *абстрактной* (`abstract`). Абстрактный характер операции указывается записью ее строки курсивом.

Если для некоторой операции необходимо дополнительно указать особенности ее реализации, например, алгоритм, то на диаграмме классов это может быть сделано в форме примечания. Подобное примечание может быть присоединено к строке операции соответствующего класса. Для представления алгоритмов реализации отдельных операций служит другой тип канонических диаграмм — диаграммы деятельности.

4.2.4. Параметр

Параметр (parameter) является спецификацией аргумента, который используется при выполнении операции или при вызове характеристики поведения.

Каждый параметр должен иметь следующий формат записи (БНФ):

```
<параметр>::= [<направление>] <имя параметра> ':' <выражение типа>
['['<кратность>']'] ['=' <значение по умолчанию>]
['{' <свойство параметра> [',' <свойство параметра>]* '}']
```

Здесь:

- <*направление*>::= **'in'** | **'out'** | **'inout'**| **'return'**. Другими словами, *направление* (direction) является одним из ключевых слов: **in**, **out**, **inout** или **return**. Если оно не указано, то по умолчанию принимается значение `in`. Семантика направления параметра описывается далее;
- <*имя параметра*> (parameter name) представляет собой идентификатор формального параметра, при записи которого необходимо следовать правилам задания имен атрибутов;
- <*выражение типа*> (type expression) является спецификацией типа данных для возможных значений соответствующего формального параметра. Этот терм аналогичен рассмотренному выше терму <*тип атрибута*> для атрибутов классов;
- <*кратность*> (multiplicity) характеризует общее количество конкретных параметров с данным именем, которые могут принадлежать тому или

иному объекту данного класса. Кратность параметра записывается в форме строки текста в соответствии с правилами, рассмотренными ранее для атрибутов классов;

- <*значение по умолчанию*> (default) представляет собой некоторое выражение, которое специфицирует конкретное значение по умолчанию для данного формального параметра;
- <*свойство параметра*> (parameter property) указывает дополнительные свойства значений данного формального параметра. В качестве значений свойств параметра могут быть использованы модификаторы атрибутов (см. табл. 4.1).

Для параметра может быть задано некоторое имя, которое в этом случае идентифицирует этот параметр единственным образом среди параметров операции или соответствующей характеристики поведения. Имя параметра, так же как и имя атрибута, начинается со строчной буквы. Если для параметра специфицируется значение по умолчанию, то оно используется в качестве аргумента для этого параметра, если и только если при вызове данной операции не применяется никакой аргумент. В качестве возвращаемого операцией результата может быть указан отдельный параметр.

Направление параметра специфицирует способ передачи его значения в операции. Вид направления параметра является типом перечисления, который определяет литералы, используемые для спецификации направления параметров. Вид направления параметра является перечислением следующих литеральных значений:

- **in** — указывает на то, что значения этого параметра передаются в операцию вызывающим объектом;
- **inout** — указывает на то, что значения этого параметра передаются в операцию вызывающим объектом и затем обратно вызывающему объекту по окончании выполнения операции;
- **out** — указывает на то, что значения этого параметра передаются вызывающему объекту по окончании выполнения операции;
- **return** — указывает на то, что значения этого параметра передаются в качестве возвращаемых значений вызывающему объекту по окончании выполнения операции.

Параметры с направлением `in` или `inout` определяют тип и число аргументов, которые должны быть предоставлены при вызове соответствующей операции. Параметр с направлением `out`, `inout` или `return` определяет тип аргумента, который должен быть возвращен после успешного вызова операции.

При этом операция может иметь не более одного возвращаемого параметра с направлением `return`.

Операция класса может быть вызвана у объекта класса, для которого данная операция специфицирована, посредством задания отдельного множества подстановок для параметров этой операции. Вызов операции может служить причиной изменения значений атрибутов этого объекта. Операция может также возвращать некоторое значение в качестве результата, если определен тип результата для этой операции. Вызовы операций также могут служить причиной изменения значений атрибутов других объектов. Вызовы операций также могут создавать и уничтожать объекты классов.

Дополнительно для операций могут быть определены предусловия и постусловия.

Предусловие (precondition) для операции определяет условие, которое должно быть истинным, когда эта операция вызывается.

Постусловие (postcondition) для операции определяет условие, которое должно быть истинным, когда вызов операции успешно завершился, в предположении, что все предусловия были удовлетворены.

Для операции может быть задано условие на тело операции, которое ограничивает значение возвращаемого результата.

Условие на тело операции отличается от постусловия тем, что условие на тело операции может быть перекрыто, когда операция переопределяется, в то время как постусловия могут быть только добавлены в ходе переопределения операции.

Ситуация, когда при вызове операции не удовлетворяется предусловие, является точкой изменения семантики. Это означает, что разработчик должен дополнительно специфицировать в модели подобную ситуацию. Операция может вызывать исключение в ходе своего вызова. Исключение не может произойти, если постусловия или условие на тело операции удовлетворены.

Операция может быть переопределена в специализации класса, который имеет спецификацию этой операции. При переопределении операции могут быть специализированы типы параметров, добавлены новые предусловия или постусловия, добавлены новые порождаемые исключения или уточнена спецификация переопределяемой операции. При переопределении операции переопределенная операция должна быть согласованной с переопределяемой

операцией. Переопределенная операция является *согласованной* с переопределяемой операцией, если она имеет то же самое число формальных параметров, такое же число возвращаемых результатов, а тип каждого формального параметра и возвращаемого результата согласуется с типом соответствующих переопределяемых параметров или возвращаемого результата.

Возвращаемый операцией результат может быть выражен или как возвращаемый параметр, или как тип операции. Например: `toString(return : String)` означает то же самое, что и `toString( ) : String`.

Можно привести следующие примеры записи операций для класса `Сотрудник`:

- `+добавить(in номерТелефона : Integer [*] {unique})` — общедоступная операция, которая использует многозначный целочисленный параметр `номерТелефона` в качестве передаваемых операций значений. Среди них не должно быть повторяющихся значений, на что указывает модификатор свойство параметра `unique`;
- `-изменить(in заработнаяПлата : Currency)` — закрытая операция с единственным передаваемым параметром `заработнаяПлата` типа `Currency`;
- `+создать() : Boolean` — общедоступная операция по созданию отдельного объекта класса `Сотрудник`, которая не содержит параметров. Эта операция может быть вызвана объектами любого класса в пространстве имен модели. После вызова данной операции она должна возвратить логическое значение.

4.3. Отношения между классами

Кроме внутреннего устройства классов важную роль при разработке моделей программных систем имеют различные отношения между классами, которые также могут быть изображены на диаграмме классов. Совокупность допустимых типов отношений, которые могут быть представлены на диаграмме классов, фиксирована в языке UML 2.0. Такими типами отношений или взаимосвязей, которые могут быть изображены на диаграммах классов, являются отношения ассоциации и обобщения, агрегации и композиции, зависимости и реализации.

Каждое из этих отношений имеет собственное графическое представление и семантику, которая отражает характер и особенности соответствующей взаимосвязи между классами.

4.3.1. Ассоциация

Ассоциация (association) — произвольное отношение или взаимосвязь между классами.

Бинарная ассоциация (binary association) обозначается сплошной линией со стрелкой или без нее, возможно, с некоторыми дополнительными символами, которые характеризуют специальные свойства ассоциации. Линии могут быть изображены с использованием различных стилей, включая ортогональные, наклонные и криволинейные сегменты. Выбор отдельного множества стилей линий предоставляется пользователю.

В качестве дополнительных специальных символов могут использоваться: имя ассоциации, символ навигации, а также имена, видимость и кратность концов ассоциации. Хотя ассоциация рассматривалась при изучении отношений диаграммы вариантов использования в *главе 3*, семантика этого типа отношений на диаграмме классов может быть использована в полном объеме.

Имя ассоциации (association name) является необязательным элементом ее обозначения. Если оно задано, то записывается с прописной (большой) буквы полужирным шрифтом и располагается рядом с символом ассоциации, но не слишком близко к ее концу, чтобы не перепутать с именем этого конца. Наиболее простой случай данного отношения — *бинарная ассоциация*, которая связывает в точности два класса. Частным случаем бинарной ассоциации является *рефлексивная* ассоциация, которая связывает класс с самим собой.

При изображении ассоциации особая роль отводится классам, расположенным на концах линии ассоциации. Соответствующий элемент в языке UML 2.0 имеет собственное название — *конец ассоциации* (association end), который графически соответствует точке соединения линии ассоциации с прямоугольником класса. При этом по умолчанию конец ассоциации представляет собой множество. Произвольная ассоциация может иметь два или больше концов ассоциации, каждый из которых, в свою очередь, может иметь имя, видимость, кратность, символ наличия или отсутствия навигации, а также строку свойства.

- *Имя конца ассоциации* специфицирует *роль* (role), которую играет класс, расположенный на соответствующем конце рассматриваемой ассоциации. Оно является необязательным, но если оно задано, то представляет собой строку текста, записывается со строчной (малой) буквы обычным шрифтом и располагается рядом с символом класса. Конец ассоциации может быть *производным*. В этом случае его имени должен предшествовать знак `/`.
- *Видимость конца ассоциации* специфицирует возможность доступа к соответствующему концу ассоциации с других ее концов. В качестве види-

мости концов ассоциации используются те же значения видимости, что для атрибутов и операций. Если она указывается совместно с именем конца ассоциации, то записывается с помощью специального символа перед этим именем.

- *Кратность конца ассоциации* специфицирует возможное количество экземпляров соответствующего класса, которое может соотноситься с одним экземпляром класса на другом конце этой ассоциации. Кратность концов ассоциации обозначается в виде интервала целых чисел, аналогично кратности атрибутов классов, но без прямых скобок.
- *Символ наличия навигации* (navigable) изображается с помощью простой стрелки в форме буквы "V" на конце ассоциации. Наличие этой стрелки указывает на то, что соответствующий класс является доступным для навигации со стороны классов на других концах ассоциации.
- *Символ отсутствия навигации* (non navigable) изображается с помощью буквы "X" на линии у конца ассоциации. Наличие этого символа указывает на то, что соответствующий класс является недоступным для навигации со стороны классов на других концах ассоциации.
- *Строка свойства* (property string) записывается в фигурных скобках и служит для указания дополнительных свойств, которые имеет соответствующий конец ассоциации. При этом набор возможных свойств конца ассоциации в языке UML 2.0 фиксирован. Назначение этих свойств описывается в табл. 4.3.

Таблица 4.3. *Назначение свойств концов ассоциаций в языке UML 2.0*

Свойство конца ассоциации	**Назначение свойства конца ассоциации**
{subsets <*имя элемента*>}	Конец ассоциации представляет собой некоторое подмножество <*имя элемента*>, в качестве которого может выступать имя конца ассоциации или атрибута класса
{redefined <*имя конца ассоциации*>}	Конец ассоциации переопределяет другой конец ассоциации с именем <*имя конца ассоциации*>
{union}	Конец ассоциации является производным и определяется посредством объединения своих подмножеств
{ordered}	Конец ассоциации представляет собой некоторое упорядоченное множество
{bag}	Конец ассоциации представляет собой мультимножество или совокупность, в которой допускается представлять один и тот же элемент более одного раза
{sequence} или {seq}	Конец ассоциации представляет собой некоторую последовательность или упорядоченное мультимножество

Отсутствие стрелки или символа отсутствия навигации означает, что навигация для соответствующего конца ассоциации не специфицирована. Стрелку наличия навигации не следует путать с символом порядка чтения классов ассоциации, который может быть указан в форме небольшого треугольника рядом с именем данной ассоциации.

Ассоциация объявляет, что между экземплярами ассоциированных классов могут существовать связи. При этом связь в языке UML 2.0, являясь экземпляром ассоциации, может быть представлена кортежем с одним значением для каждого конца ассоциации. Кратность конца ассоциации ограничивает размер этой коллекции. Если конец ассоциации помечен как упорядоченный, то коллекция будет упорядоченной. Если дополнительно конец ассоциации помечен как уникальный, то эта коллекция является множеством; в противном случае она допускает повторение элементов.

В качестве примера бинарной ассоциации рассмотрим отношение между двумя классами — классом `Компания` и классом `Сотрудник` (рис. 4.3). Они связаны между собой бинарной ассоциацией с именем `Работает`, которое указано на рисунке рядом с линией ассоциации. Для данного отношения определен порядок чтения ассоциации, а именно: `Сотрудник Работает в Компании`. Навигация для концов данной ассоциации не специфицирована. Специальные строки свойств для концов ассоциации здесь отсутствуют.

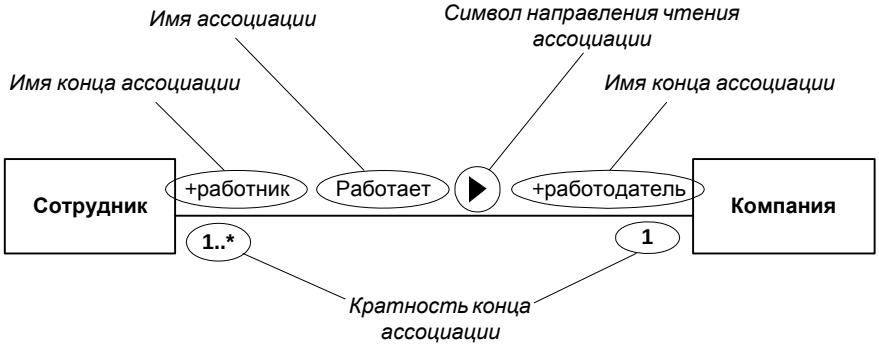


Рис. 4.3. Графическое изображение ненаправленной бинарной ассоциации между классами

Конец ассоциации класса `Сотрудник` (см. рис. 4.3) имеет имя `работник`, общедоступную видимость и кратность `1..*`. Конец ассоциации класса `Компания` имеет имя `работодатель`, общедоступную видимость и кратность `1`. Наличие указанных кратностей будет означать, что в рамках рассматриваемой модели любой конкретной компании может соответствовать несколько конкретных

сотрудников. Это может быть легко интерпретировано в форме: в компании работают несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено.

С другой стороны, любому трудоустроенному сотруднику будет всегда соответствовать единственная компания. Это означает, что в рамках рассматриваемой модели недопустимо одновременная работа сотрудников в нескольких компаниях. Заметим, что если вместо кратности `1..*` записать символ `*`, то изменится семантика этого элемента модели. А именно, поскольку последний символ эквивалентен кратности `0..*`, то для данной модели это означало бы, что отдельные компании могут совсем не иметь сотрудников в своем штате.

Понять назначение концов ассоциации помогает знание особенностей генерации программного кода на основе диаграммы классов. Если для некоторого конца ассоциации указаны его имя, видимость, кратность и навигация на противоположном конце, то при генерации программного кода для класса будет добавлен новый атрибут с именем рассматриваемого конца ассоциации, имеющий соответствующую видимость и кратность. При этом в качестве типа нового атрибута будет использовано имя класса на противоположном конце рассматриваемой ассоциации. Если же для противоположного конца ассоциации не указан символ навигации, то при генерации программного кода для класса никакого нового атрибута добавлено не будет.

В качестве примера ассоциации со специфицированной навигацией рассмотрим отношение между двумя классами — классом `Многоугольник` и классом `Линия` (рис. 4.4, *а*). Они связаны между собой бинарной ассоциацией `Содержит`, для которой указана навигация на конце ассоциации с именем `сторона` и отсутствует навигация на противоположном конце ассоциации. Это означает, что класс `Многоугольник` должен содержать информацию о своих сторонах. Данный факт может быть представлен альтернативным образом в форме класса `Многоугольник` с атрибутом `сторона`, для которого в качестве типа атрибута используется класс `Линия` (рис. 4.4, *б*).

Рис. 4.4. Графическое изображение: *а* — бинарной ассоциации с навигацией; *б* — эквивалентное ему представление класса с атрибутом

Рассмотренный пример иллюстрирует то обстоятельство, что конец ассоциации является частью класса, а не ассоциации. В то же время во многих CASE-средствах концы ассоциации специфицируются в свойствах ассоциации, о чем следует помнить разработчикам при использовании соответствующих программ.

Нотация языка UML 2.0 допускает различные комбинации наличия и отсутствия навигации у концов ассоциации (рис. 4.5).

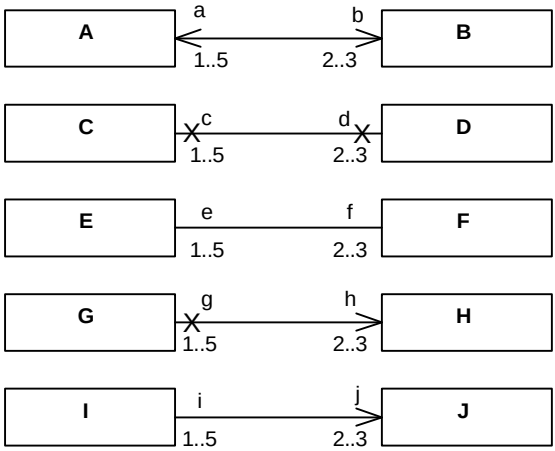


Рис. 4.5. Различные варианты изображения навигации и кратности у концов ассоциации

Здесь все изображенные классы попарно связаны бинарными ассоциациями. При этом концы ассоциаций для классов A, B, H, J имеют навигацию. Концы ассоциаций для классов C, D, G не имеют навигации. Для концов ассоциаций у классов E, F, I навигация не специфицирована.

Частным случаем отношения ассоциации является так называемая *исключающая ассоциация* (Xor-association). Семантика данной ассоциации указывает на тот факт, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент времени может использоваться только один ее экземпляр. На диаграмме классов исключающая ассоциация изображается с помощью дополнительной пунктирной линии, соединяющей две и более ассоциации, рядом с которой записывается ограничение в форме специального ключевого слова `{xor}`.

Например, предположим, что для некоторой модели счет в банке может быть открыт для любого клиента, в качестве которого может выступать либо

физическое лицо (индивидуум), либо юридическое лицо (компания). Эта ситуация на диаграмме классов может быть представлена следующим образом. Базовый класс `Счет в банке` связывается исключающей ассоциаций с классом `Физическое лицо` и с классом `Юридическое лицо` (рис. 4.6).

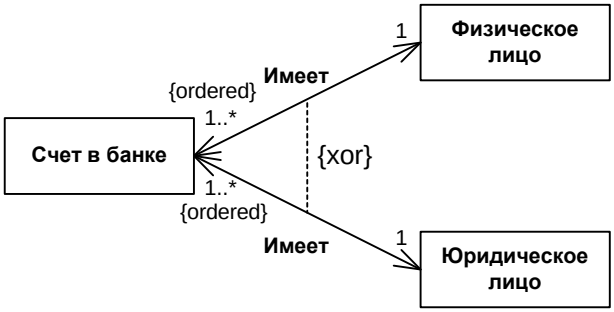


Рис. 4.6. Графическое изображение исключающей ассоциации между тремя классами

Все концы данной ассоциации должны иметь навигацию, которая является необходимой для установления взаимного соответствия между конкретными счетами в банке и конкретными клиентами. Кратность концов ассоциации устанавливает допустимое количество экземпляров классов в конкретных реализациях ассоциации. В дополнение к этому конкретные счета как физических, так и юридических лиц являются упорядоченными, например, по дате их открытия. Это требование специфицируется с помощью строки свойства: `{ordered}` (см. табл. 4.3).

Если две линии ассоциаций на диаграмме не пересекаются, отсутствие пересечения может быть показано небольшой полукруглый дугой (как на диаграммах электрических цепей). Для изображения стрелок навигации на диаграммах классов могут быть выбраны различные опции. На практике часто бывает удобно скрыть некоторые из стрелок или символов отсутствия навигации, о чем разработчику необходимо дополнительно указать в документации. Наиболее часто используются следующие соглашения:

- показываются все стрелки и символы отсутствия навигации. При этом навигацию и ее отсутствие изображают только явным образом (см. рис. 4.5);
- скрываются все стрелки и символы отсутствия навигации. В этом случае о навигации нельзя сделать никакого вывода, что аналогично ситуации, в которой информация в представлении скрыта. Этот случай на практике встречается достаточно редко;

- скрываются стрелки для ассоциации с направленностью в обоих направлениях и показываются стрелки только для ассоциации с односторонней направленностью. В этом случае двусторонняя направленность не отличается от ситуации, когда навигации не существует вовсе, и этот случай на практике встречается достаточно часто.

4.3.2. *N*-арная ассоциация

Более общий случай представляет собой *n-арная ассоциация* (n-ary association, читается — "эн арная ассоциация"), которая связывает некоторым отношением ассоциации более двух классов. В качестве значения *N* может выступать произвольное натуральное число больше 1. Каждый экземпляр реализации *n*-арной ассоциации представляет собой *n*-арный кортеж, состоящий в точности из *n* объектов соответствующих классов. Частным случаем *n*-арной ассоциации является *тернарная ассоциация*, которая связывает соответствующим отношением в точности три класса ($n = 3$). Бинарная ассоциация также является частным случаем *n*-арной ассоциации ($n = 2$), хотя и имеет свое собственное обозначение. Концы *n*-арной ассоциации для $n \geq 3$ являются собственностью этой ассоциации.

N-арная ассоциация графически обозначается *ромбом*, от которого ведут линии к символам классов данной ассоциации. Ромб соединяется с символами соответствующих классов сплошными линиями, которые проводятся от вершин или от середины сторон. Имя *n*-арной ассоциации записывается рядом с ромбом соответствующей ассоциации. Однако порядок классов в *n*-арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме классов не фиксируется.

Вообще говоря, любая ассоциация может быть изображена в виде ромба, форма которого больше, чем терминатор на линии агрегации, и с непрерывными линиями для каждого конца ассоциации, связывающими этот ромб с классом, который является типом этого конца. Ассоциация с более чем двумя концами может быть изображена только таким способом.

В качестве примера 4-арной ассоциации рассмотрим отношение `Игра` между классами: `Футбольная команда`, `Год` и `Дата`. Данная ассоциация может представлять информацию о состоявшихся играх футбольных команд в национальном чемпионате в течение нескольких лет (рис. 4.7).

Отдельным экземпляром ассоциации `Игра` может служить, например, кортеж `<Динамо, Зенит, 17 июля, 2005>`. В рассматриваемой модели вполне возможно, что 2 футбольные команды не провели в конкретном сезоне между собой ни одной встречи, на что указывают соответствующие кратности концов

ассоциации. Все состоявшиеся игры могут быть упорядочены по годам и дате, на что указывает строка свойства `{ordered}` у соответствующих концов ассоциации.

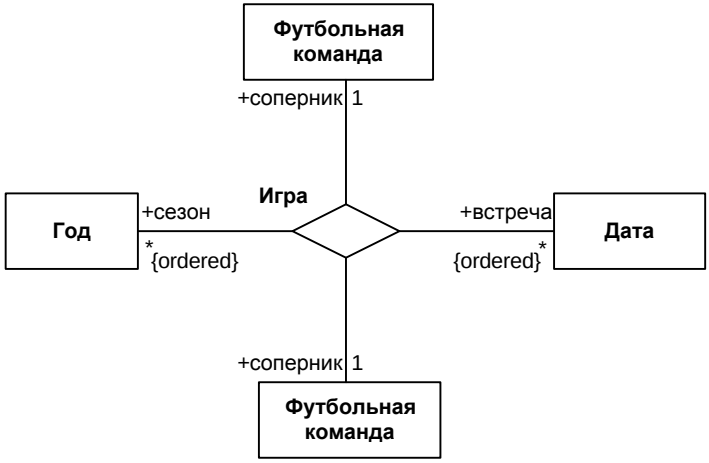


Рис. 4.7. Графическое изображение 4-арной ассоциации

Фрагмент диаграммы классов (см. рис. 4.7) обладает очевидным недостатком — изображенная ассоциация не позволяет представить информацию ни о месте состоявшейся встречи, ни о ее результате. Для этой цели в языке UML 2.0 может быть использована специальная конструкция, получившая название "ассоциация-класс".

4.3.3. Ассоциация-класс

Ассоциация-класс (association class) — элемент модели, который имеет свойства как ассоциации, так и класса, и предназначенный для спецификации дополнительных свойств ассоциации в форме атрибутов и, возможно, операций класса.

Ассоциация-класс может рассматриваться как ассоциация, которая также имеет свойства класса, или как класс, который также имеет свойства ассоциации. Она не только соединяет множество классов, но и определяет множество характеристик, которые принадлежат самому отношению и не принадлежат ни одному из классов.

Семантика ассоциации-класса является комбинацией семантики обычной ассоциации и класса, поскольку в языке UML 2.0 ассоциация-класс является

примером множественного наследования от метаклассов `Ассоциация` и `Класс`. Это означает, что ассоциация-класс является одновременно видом ассоциации и видом класса. Обе эти конструкции являются классификаторами и, следовательно, имеют множество общих свойств, таких как способность иметь характеристики, иметь имя и т. д. Поскольку эти свойства наследуются от конструкции `Классификатора`, они не повторяются. Поэтому ассоциация-класс имеет только одно имя и множество характеристик, которые определены для классов и для ассоциаций. Ограничения, определенные для класса и для ассоциации, также применимы для ассоциации-класса, из чего следует, например, что атрибуты ассоциации-класса, концы ассоциации-класса и противоположные концы ассоциации, соединенные с ассоциацией-классом, должны иметь все различные имена. Более того, правила специализации и уточнения, определенные для класса и ассоциации, являются также применимыми для ассоциации-класса.

Ассоциация-класс изображается в форме символа класса, присоединенного к пути ассоциации посредством пунктирной линии. Путь ассоциации и символ ассоциации-класса представляют один и тот же элемент модели, который имеет отдельное имя. Имя может быть помещено на пути ассоциации, в символе класса или на обоих, но это должно быть одно и то же имя. Логически ассоциация-класс и ассоциация являются одной и той же семантической сущностью, однако графически они различаются. Атрибуты и операции ассоциации-класса указываются в соответствующих секциях прямоугольника класса. Ассоциация-класс может быть использована для расширения свойств *N*-арной ассоциации.

В качестве примера преобразуем рассмотренную ранее 4-арную ассоциацию `Игра` в ассоциацию класс, что позволит представлять информацию о результате и месте состоявшейся встречи (рис. 4.8).

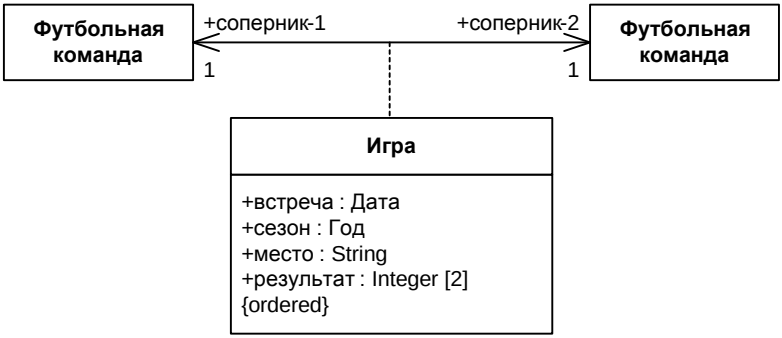


Рис. 4.8. Графическое изображение ассоциации класса

В этом случае отдельным экземпляром ассоциации класса `Игра` может служить, например, информация о конкретно состоявшейся встрече, записанной в форме кортежа: `<Динамо, Зенит, игра : Игра>`. При этом для объекта `игра : Игра` задаются конкретные значения атрибутов: `встреча = 17 июля`, `сезон = 2005`, `место = 'Москва'`, `результат = <1, 2>`. Поскольку значения результата упорядочены по соперникам, то в более привычной нотации они могут быть записаны в виде: `'1 : 2'`. Заметим, что правила обозначения объектов или экземпляров классов будут рассмотрены далее в книге.

Следует заметить, что в экземпляре ассоциации класса существует только один экземпляр ассоциированных классов на каждом из концов, т. е. с точки зрения экземпляра кратность концов ассоциации класса равна 1.

4.3.4. Квалификатор

Квалификатор (qualifier) объявляет разбиение множества ассоциированных экземпляров относительно экземпляра на квалифицированном конце ассоциации. Экземпляр квалификатора состоит из одного значения для атрибута каждого квалификатора. Если задан некоторый квалифицированный объект и экземпляр квалификатора, число объектов на другом конце ассоциации ограничено объявленной кратностью. В случае, когда кратность равна `0..1`, значение квалификатора является уникальным относительно квалифицированного объекта и обозначает не более одного ассоциированного объекта. В случае, когда кратность равна `0..*`, множество ассоциированных экземпляров разбивается на подмножества, каждое из которых соответствует заданному экземпляру квалификатора. В случае, когда кратность равна `1` или `0..1`, квалификатор имеет последствия как для семантики, так и для реализации. В случае, когда кратность равна `0..*`, это не имеет реальных семантических последствий, но предлагает реализацию, которая должна обеспечить простой доступ ко множеству ассоциированных экземпляров, связанных с данным значением квалификатора.

Квалификатор изображается в форме маленького прямоугольника, присоединенного к концу пути ассоциации между сегментом окончания пути и символом класса (рис. 4.9, *а*) или к стрелке композиции, изображенной в форме ромба (рис. 4.9, *б*). При этом прямоугольник квалификатора должен быть меньше, чем соответствующий прямоугольник класса, хотя это не всегда практично. Прямоугольник квалификатора является частью пути ассоциации, но не частью класса. Квалификатор присоединяется к концу ассоциации источника. Кратность, присоединенная к концу цели, обозначает возможные мощности множеств экземпляров цели, попарно выбранные от экземпляра

источника к значению квалификатора. Атрибуты квалификатора изображаются в прямоугольнике квалификатора, при этом один или более атрибутов могут быть изображены на одной линии. Атрибуты квалификатора имеют такую же нотацию, что и атрибуты класса, за исключением того, что выражения для начального значения не имеют смысла. Квалификатор не может быть скрыт.

а *б*

Рис. 4.9. Графическое изображение квалификатора:
а — для ассоциации; *б* — для композиции

Ассоциация является наиболее общей формой отношения в языке UML 2.0. Практически все другие типы отношений в языке UML 2.0 являются специальными случаями данного отношения. Эти отношения относятся к базовым понятиям языка UML 2.0, имеют собственную семантику и специальные обозначения, что требует самостоятельного рассмотрения.

4.3.5. Обобщение

Обобщение (generalization) — таксономическое отношение между более общим классификатором (родителем или предком) и более специальным классификатором (дочерним или потомком).

Семантика отношения обобщения означает, что каждый экземпляр специального классификатора также является экземпляром общего классификатора. Поэтому характеристики, специфицированные для экземпляров общего классификатора, являются неявно специфицированными для экземпляров специального классификатора. В этом случае говорят, что специальный классификатор *наследует* характеристики более общего классификатора. Любое ограничение, накладываемое на экземпляры общего классификатора, также применимо к экземплярам специального классификатора.

Отношение обобщения является направленным и может использоваться для представления иерархических взаимосвязей между классами, вариантами использования и другими элементами модели, которые являются классификаторами в нотации языка UML 2.0.

Согласно одному из главных принципов методологии ООАП — наследованию, класс-потомок обладает всеми свойствами и поведением класса-предка, а также может иметь дополнительные свойства и поведение, которые отсутствуют у класса-предка. Применительно к диаграмме классов обобщение обеспечивает наследование атрибутов и операций классов и обозначается сплошной линией со стрелкой в форме не закрашенного треугольника на одном из своих концов (рис. 4.10). Стрелка указывает на общий класс или класс-предок, а ее противоположный конец — на специальный класс или класс-потомок (рис. 4.10, *а*).

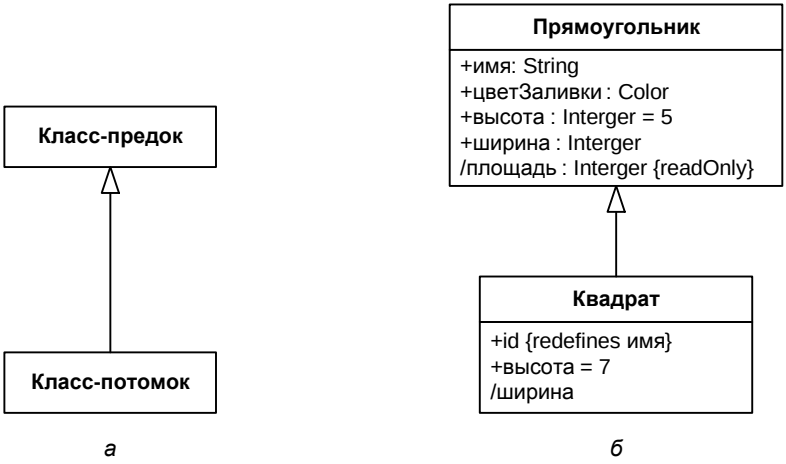


Рис. 4.10. Графическое изображение отношения обобщения в языке UML 2.0: *а* — свернутое представление классов; *б* — развернутое представление классов

Для изображенного справа отношения обобщения (рис. 4.10, *б*) класс `Квадрат` наследует от класса `Прямоугольник` все атрибуты. Дополнительный атрибут `id` класса `Квадрат` переопределяет атрибут `имя` класса `Прямоугольник`, а атрибут `высота` класса `Квадрат` имеет по умолчанию значение `7`, вместо значения `5` в родительском классе. Атрибут `ширина` в классе `Квадрат` является производным, поскольку его значение всегда равно значению атрибута `высота`. Хотя атрибуты `цвет Заливки` и `площадь` в классе `Квадрат` не указаны, они также принадлежат этому классу. Их спецификация полностью идентична описанию в родительском классе, что следует из семантики отношения обобщения.

Отношение обобщения, которое называют также отношением классификации или наследования, допускает, чтобы от одного класса-предка одновременно наследовали несколько классов-потомков. Это обстоятельство отражает таксономический характер данного отношения. В языке UML 2.0 допускается также, чтобы класс-потомок наследовал от нескольких классов-предков (так называемое *множественное наследование*). Во всех этих случаях на диаграмме классов указывается несколько линий обобщения с соответствующими стрелками.

Например, класс *Геометрическая фигура* (курсив обозначает абстрактный класс) может выступать в качестве родительского класса для классов-потомков, соответствующих конкретным геометрическим фигурам, таким как Прямоугольник, Окружность, Эллипс и др. Данный факт может быть представлен графически в форме следующей диаграммы классов (рис. 4.11).

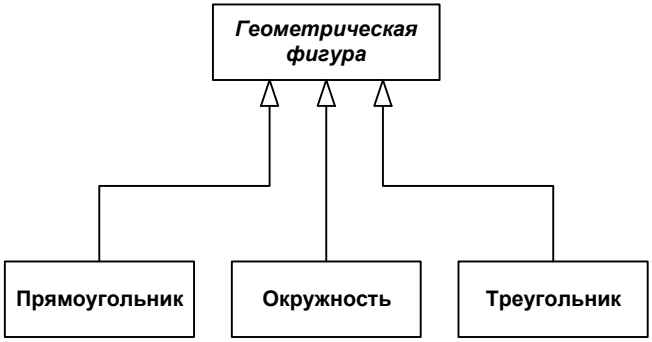


Рис. 4.11. Пример графического изображения отношения обобщения для нескольких классов-потомков

Несколько отношений обобщения, которые ссылаются на один и тот же общий классификатор, могут быть соединены совместно в форме "стиля совместно используемой цели". В этом случае отдельные линии изображаются сходящимися к единственной стрелке, которая имеет с этими линиями единственную точку пересечения (рис. 4.12). Этот стиль наиболее часто используется при обозначении множества обобщения, рассматриваемого далее.

Графическое изображение отношения обозначения по форме соответствует графу специального вида, а именно иерархическому дереву. Как нетрудно заметить, в этом случае класс-предок является корнем дерева, а классы-потомки — его листьями. Отличие заключается в возможности указания на диаграмме классов дополнительной семантической информации, которая может отражать различные теоретико-множественные характеристики

данного отношения. При этом класс-предок на диаграмме может занимать произвольное положение относительно своих классов-потомков, что определяется соображениями удобства.

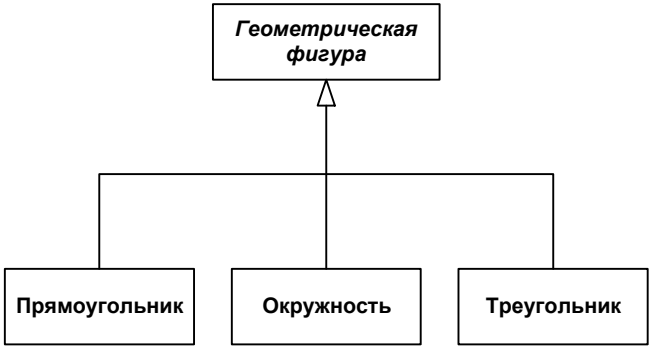


Рис. 4.12. Альтернативный вариант графического изображения отношения обобщения классов (см. рис. 4.11) в "стиле совместно используемой цели"

Иерархии обобщения всегда должны быть ациклическими. Другими словами, классификатор не может быть одновременно транзитивно общим и транзитивно специальным классификатором одного и того же классификатора. Невыполнение этого условия приводит к несогласованной диаграмме.

Обобщения между ассоциациями могут быть изображены с использованием стрелки обобщения между символами ассоциации. Обобщения между ассоциациями лучше изображать с помощью линии другого цвета или ширины, чем та, которая используется для ассоциации.

4.3.6. Множество обобщения

Множество обобщения (generalization set) — элемент модели, экземпляры которого определяют коллекции подмножеств отношения обобщения.

Каждое множество обобщения определяет отдельное множество отношений обобщения, которые описывают способ, которым общий классификатор (предок) может быть разделен на несколько специальных подтипов. При этом каждая коллекция подмножеств представляет некоторое ортогональное измерение специализации общего классификатора.

Например, можно определить множество обобщения как разбиение класса *Человек* на два класса: Мужчина и Женщина. Здесь множество обобщения следует ассоциировать с двумя экземплярами обобщения, которые должны иметь *Человек* в качестве общего класса. Однако одно отношение обобщения предполагает, что Мужчина является специальным классом, а другое отношение предполагает, что Женщина является специальным классом. Другими словами, можно сказать, что класс *Человек* разбивается на два подкласса: Мужчина и Женщина.

Класс *Человек* может быть также классифицирован по профессиям на классы: Программист, Руководитель Проекта и Бизнес-аналитик. Эта коллекция подмножеств будет определять другое множество обобщения, которое ассоциируется с тремя другими отношениями обобщения. Все они также должны иметь *Человек* в качестве общего класса. При этом эти специальные классы будут отличаться от классов Мужчина и Женщина из другого множества обобщения. Соответствующая диаграмма классов может быть изображена так, как представлено на рис. 4.13.

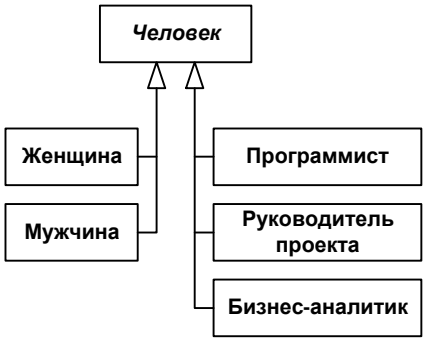


Рис. 4.13. Диаграмма классов для двух множеств обобщения

В дополнение к сказанному следует заметить, что *Человек* является абстрактным классом, поскольку объект *Человек* не может существовать без того, чтобы быть или Женщиной, или Мужчиной.

Множество обобщения может содержать строку текста, указывающую на некоторые специальные свойства этого отношения. Этот текст будет относиться ко всем линиям обобщения, которые идут к классам-потомкам данного множества обобщения. Чтобы указать, покрывает или нет множество обобщения, а также пересекаются или нет его элементы, каждое множество должно быть помечено одним из следующих ограничений:

- `{complete, disjoint}` — означает, что данное множество обобщения является покрывающим и его специальные классы не имеют общих экземпляров;

- ❒ `{incomplete, disjoint}` — означает, что данное множество обобщения не является покрывающим и его специальные классы не имеют общих экземпляров (предполагается по умолчанию);
- ❒ `{complete, overlapping}` — означает, что данное множество обобщения является покрывающим и его специальные классы имеют общие экземпляры;
- ❒ `{incomplete, overlapping}` — означает, что данное множество обобщения не является покрывающим и его специальные классы имеют общие экземпляры.

Графически ограничения на множества обобщения изображаются с использованием нотации общей стрелки обобщения (рис. 4.14, *а*) или пунктирной линии для отдельного множества обобщения (рис. 4.14, *б*).

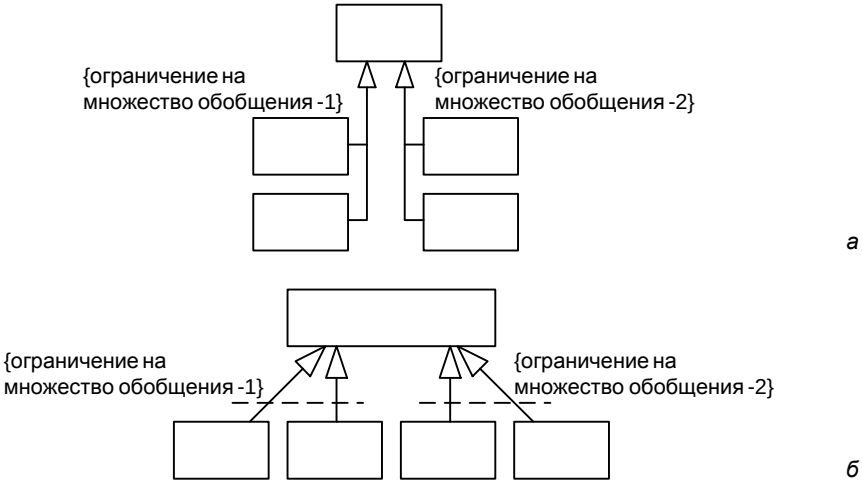


Рис. 4.14. Варианты нотации для ограничения на множество обобщения: *а* — с помощью общей стрелки обобщения; *б* — с помощью пунктирной линии

В рассмотренном ранее примере класс *Человек* в первом множестве обобщения может быть специализирован или как Женщина, или как Мужчина. Поскольку это множество обобщения разбивается, каждый экземпляр *Человек* должен быть или Женщиной, или Мужчиной, т. е. он должен быть или одним, или другим, но не обоими. При этом других классов данное множество обобщения не содержит. Таким образом, здесь имеет место ограничение: `{complete, disjoint}`.

Класс *Человек* во втором множестве обобщения может быть специализирован как Программист, Руководитель Проекта и Бизнес-аналитик. Это означает, что

экземпляры *Человека* могут быть подмножеством Программиста, Руководителя Проекта и Бизнес-аналитика или некоторой другой коллекции, которая содержит экземпляры, не относящиеся к указанным трем классам. Тем самым это отношение представляет неполную классификацию. Поскольку в некоторых случаях программист или бизнес-аналитик могут одновременно выполнять обязанности руководителя проекта, соответствующие подмножества экземпляров этих классов не будут являться непересекающимися. Таким образом, здесь имеет место ограничение: {incomplete, overlapping}.

С учетом использования ограничений на множество обобщения изображенная ранее диаграмма классов (см. рис. 4.13) может быть уточнена и специфицирована следующим образом (рис. 4.15).

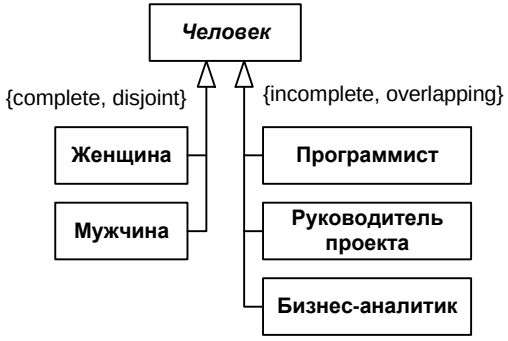


Рис. 4.15. Вариант графического изображения отношения обобщения классов с использованием ограничений на множество обобщения

4.3.7. Агрегация

Агрегация (aggregation) — направленное отношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности.

Агрегация имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое". Раскрывая внутреннюю структуру системы, агрегация показывает, из каких элементов состоит система, и как они связаны между собой. С точки зрения модели отдельные части системы могут выступать как в виде элементов, так и в виде подсистем, которые, в свою очередь, тоже могут состоять из подсистем или элементов.

Агрегация является частным случаем отношения ассоциации, от которой она наследует такие свойства, как имена концов ассоциации, кратность и ограничения. Однако агрегациями могут быть только бинарные ассоциации. При этом семантика данного отношения по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые при необходимости также могут быть подвергнуты декомпозиции.

В связи с рассмотрением данного отношения вполне уместно вспомнить о специальном термине "агрегат", который служит для обозначения технической системы, состоящей из взаимодействующих составных частей или подсистем. Эта аналогия не случайна и может служить для более наглядного понимания сути рассматриваемого отношения.

Рассматриваемое в таком аспекте деление системы на составные части также представляет собой некоторую иерархию, однако данная иерархия принципиально отличается от той, которая порождается отношением обобщения. Отличие заключается в том, что части системы имеют другой тип сущности, а значит, не наследуют ее свойства и поведение, поскольку являются вполне самостоятельными сущностями. Более того, части целого обладают своими собственными атрибутами и операциями, которые отличаются от атрибутов и операций целого.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой незакрашенный внутри (пустой) ромб в качестве дополнительного обозначения терминала на агрегированном конце линии ассоциации. Ромб агрегации по форме должен быть меньше, чем ромб в качестве нотации ассоциации. Этот ромб указывает на тот из классов, который представляет собой "класс-целое" или класс-контейнер. Остальные классы являются его "частями" (рис. 4.16).

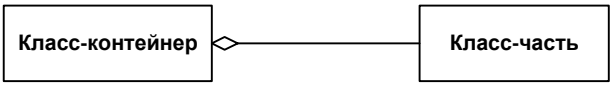


Рис. 4.16. Графическое изображение отношения агрегации

Если существуют две или более агрегации для одного агрегата, они могут быть изображены в форме дерева посредством слияния концов агрегации в отдельный сегмент. Любые дополнительные обозначения для этого отдельного сегмента применяются ко всем концам этой агрегации. Примером отношения агрегации может служить известное каждому из читателей разделение персонального компьютера на составные части: системный блок, монитор, клавиатуру и мышь. Используя обозначения языка UML 2.0, компо-

нентный состав ПК можно представить в виде соответствующей диаграммы классов (рис. 4.17), которая в данном случае иллюстрирует отношение агрегации.

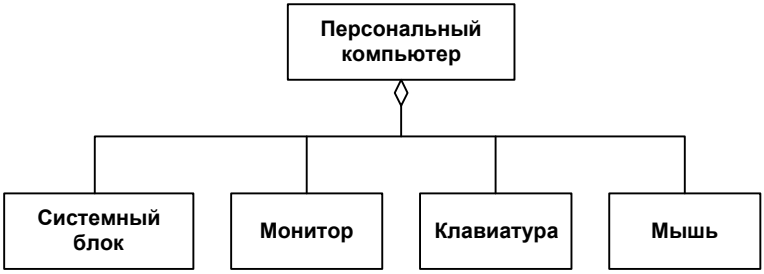


Рис. 4.17. Диаграмма классов для иллюстрации отношения агрегации на примере ПК

Точный жизненный цикл объектов класса-контейнера и классов-частей для отношения агрегации в языке UML 2.0 является точкой изменения семантики, что требует от разработчика дополнительной спецификации соответствующей информации в модели.

4.3.8. Композиция

Композиция (composition) или *композитная агрегация* предназначена для спецификации более сильной формы отношения "часть-целое", при которой с уничтожением объекта класса-контейнера уничтожаются и все объекты, являющиеся его составными частями.

Отношение композиции является частным случаем отношения агрегации. При этом она является сильной формой агрегации, которая требует, чтобы экземпляр-часть был включен не более чем в один агрегат или *композит*. Если композит удаляется, все его части обычно удаляются вместе с ним. В языке UML 2.0 объект-часть при необходимости может быть удален из композита до того, как сам композит будет уничтожен.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой ромб, который имеет заливку ("черный ромб"). Этот ромб указывает на тот из классов, который представляет собой класс-композит. Остальные классы являются его "частями" (рис. 4.18).

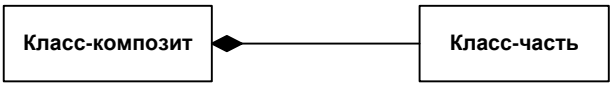


Рис. 4.18. Графическое изображение отношения композиции

Возможно, самый наглядный пример этого отношения представляет собой живая клетка в биологии, в отрыве от которой не могут существовать ее составные части. Другой пример — окно графического интерфейса программы, которое может состоять из строки заголовка, кнопок управления размером, полос прокрутки, главного меню, рабочей области и строки состояния. Подобное окно представляет собой класс, а его элементы также являются классами.

Разработчикам необходимо помнить, что агрегация и композиция представляют собой транзитивные асимметричные отношения, откуда следует обязательное требование — их связи образуют направленный ациклический граф.

Для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно могут указываться кратности отдельных классов, которые в общем случае не являются обязательными. Однако кратность агрегированного конца композитной агрегации не должна иметь верхней границы больше чем 1. Применительно к описанному ранее примеру класс `Окно программы` является классом-композитом, а взаимосвязи составляющих его частей могут быть изображены следующей диаграммой классов (рис. 4.19).

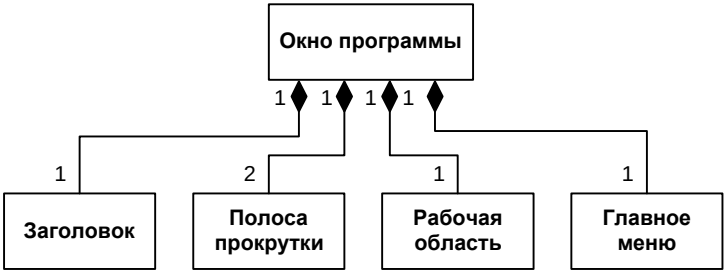


Рис. 4.19. Диаграмма классов для иллюстрации отношения композиции на примере класса-композита `Окно программы`

Данный пример иллюстрирует и другие особенности разрабатываемой компьютерной программы, которые явно не указывались в тексте этого примера. Так, в частности, указание кратности 1 рядом с классом `Рабочая область` может характеризовать одно документное приложение. В языке UML 2.0 порядок и способ, которым экземпляры части создаются в композите, не определены.

4.3.9. Зависимость

Зависимость (dependency) — отношение, предназначенное для описания ситуации, когда отдельному элементу или множеству элементов модели требуются другие элементы модели для своей спецификации или реализации.

Зависимость означает отношение типа "поставщик/клиент" между элементами модели, когда модификация поставщика может оказывать влияние на одного или несколько клиентов. *Клиент* — элемент или элементы модели, зависимые в некотором контексте от элемента или элементов *поставщика*. Из отношения зависимости следует то, что семантика клиента не является полной без поставщика. Это означает, что полная семантика зависимых элементов семантически или структурно зависит от определения элемента или элементов поставщика. Наличие отношения зависимости в модели не имеет никакой семантики времени выполнения, поскольку задается в терминах элементов модели, которые принимают участие в этом отношении, а не в терминах их экземпляров.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов. На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости или зависимого класса к классу-поставщику или независимому классу (рис. 4.20).

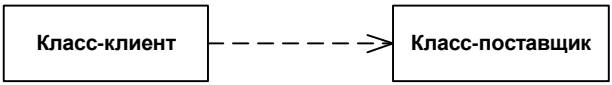


Рис. 4.20. Графическое изображение отношения зависимости на диаграмме классов

Частные случаи отношения зависимости были рассмотрены ранее *(см. главу 4)*. Линия зависимости может быть помечена необязательным стереотипом и необязательным именем. Возможно наличие нескольких элементов для клиента или поставщика. В этом случае одна или более стрелок от клиентов соединяются с одной или более стрелкой, указывающих на поставщиков. В месте соединения может быть размещена небольшая точка, если в этом есть необходимость. К точке соединения можно присоединить примечание для уточнения этой зависимости.

На практике указание направления зависимости, т. е. назначение элемента клиентом, находится в ведении разработчика и является условием соглашения. В общем случае сами разработчики могут оговаривать смысл зависимости

в соответствии с их предметной областью. На диаграммах классов зависимость и ее частный случай — отношение реализации — наиболее часто используются в контексте интерфейсов.

4.3.10. Реализация

Реализация (realization) — специализированное отношение зависимости между двумя элементами модели, один из которых представляет некоторую спецификацию *(поставщик)*, а другой представляет его реализацию *(клиент)*.

Отношение реализации означает, что множество элементов клиента является реализацией множества элементов поставщика, которое служит в качестве спецификации. Зависимость реализации изображается в форме пунктирной линии с треугольной стрелкой на конце, которая соответствует реализуемому элементу. Так, например, изображенный на рис. 4.21 фрагмент иллюстрирует случай, когда класс `Бизнес` реализуется комбинацией классов `Владелец` и `Сотрудник`.

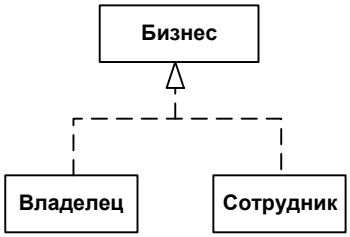


Рис. 4.21. Графическое изображение отношения реализации на диаграмме классов

4.4. Интерфейс

Интерфейс (interface) — вид класса, который представляет собой объявление множества общедоступных характеристик и обязанностей.

Интерфейс в языке UML 2.0 является специальным случаем класса, у которого, как правило, имеются операции и отсутствуют атрибуты. Интерфейсы предназначены для спецификации таких операций класса, объявления которых видимы извне, однако особенности их реализации скрыты от клиентов.

В этом смысле интерфейс не может существовать отдельно от класса, который реализует объявленные в нем операции. Если в интерфейсе объявлен атрибут, то это вовсе не означает, что реализующий экземпляр класса должен обязательно иметь этот атрибут в своей реализации, а только то, что этот атрибут будет представляться таким для внешних наблюдателей. Поскольку интерфейс является только лишь объявлением, он не является инстанцируемым элементом модели, т. е. во время выполнения экземпляры интерфейсов не существуют.

Интерфейс специфицирует некоторый контракт. Любой экземпляр класса, который реализует интерфейс, должен выполнять этот контракт. Обязанности, которые могут быть ассоциированы с интерфейсом, могут быть заданы в форме различных видов ограничений или спецификаций протоколов, которые накладывают ограничения на особенности взаимодействия с интерфейсом. Спецификация интерфейса реализуется посредством экземпляра некоторого инстанцируемого класса. Следует заметить, что в общем случае один класс может реализовывать более одного интерфейса и один интерфейс может быть реализован несколькими различными классами.

На диаграмме классов интерфейс изображается в виде маленького круга, рядом с которым записывается его имя. В качестве имени может быть существительное, которое характеризует соответствующую информацию или сервис (например, `Датчик температуры`, `Форма ввода`, `Видеокамера`). С учетом языка реализации модели имя интерфейса, как и имена других классов, рекомендуется записывать на английском. В этом случае оно должно начинаться с заглавной буквы `I`, например, `ITemperatureSensor`, `ISecureInformation`.

Множество интерфейсов, реализуемых классификатором, являются его *предоставляемыми интерфейсами* (provided interface), описывающие обязанности или сервисы, которые экземпляры этого классификатора имеют перед своими клиентами. Соответствующее отношение между классом и предоставляемым им интерфейсом представляется отношением реализации интерфейса от класса к интерфейсу. Реализация интерфейса является специализированным отношением реализации между классом и интерфейсом. Это отношение означает, что реализующий класс согласуется с контрактом, специфицированным этим интерфейсом. Оно изображается представлением интерфейса в форме круга или шара, помеченного именем интерфейса, присоединенного непрерывной линией к классу, который реализует этот интерфейс (рис. 4.22, справа).

Интерфейсы могут быть также использованы в качестве *требуемых интерфейсов* (required interface), которые специфицируют сервисы, необходимые классу для того, чтобы выполнять собственные обязанности для своих клиентов.

Соответствующее отношение между классом и требуемым интерфейсом представляется отношением зависимости использования от класса к интерфейсу. Оно изображается представлением интерфейса в форме полукруга, помеченного именем интерфейса, присоединенного непрерывной линией к классу, который требует этот интерфейс (рис. 4.22, слева).

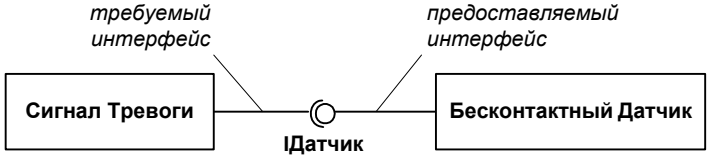


Рис. 4.22. Пример графического изображения требуемого *(слева)* и предоставляемого *(справа)* интерфейса

В данном примере `IДатчик` является предоставляемым интерфейсом для `Бесконтактного Датчика` и требуемым интерфейсом для `Сигнала Тревоги`.

Нотация предоставляемого и требуемого интерфейса в форме круга и полукруга получила свое собственное жаргонное название — "леденец на палочке", которое стало своеобразной визитной карточкой нововведений языка UML 2.0.

Как классификатор, интерфейс может быть также изображен с использованием символа прямоугольника с ключевым словом `<<interface>>`, записанным в форме стереотипа. В случае, когда интерфейс представляется с использованием нотации прямоугольника, отношения зависимости реализации и использования интерфейса обозначаются стрелками соответствующей зависимости (рис. 4.23).

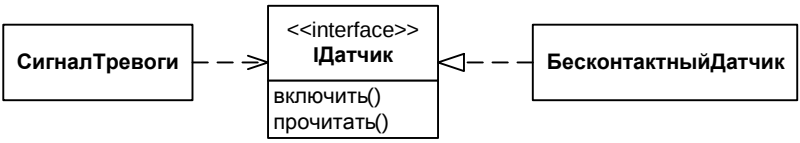


Рис. 4.23. Пример альтернативной нотации для графического изображения интерфейса для ситуации, изображенной на рис. 4.22

С системно-аналитической точки зрения интерфейс не только отделяет спецификацию операций системы от их реализации, но и задает общие границы проектируемой системы. В последующем интерфейс может быть уточнен явным указанием тех операций, которые специфицируют отдельный аспект

поведения системы. Графическое изображение интерфейсов в форме круга и полукруга также используется на других канонических диаграммах, например, на диаграммах композитной структуры и диаграммах компонентов.

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом. Более того, спецификация интерфейсов способствует "безболезненной" модификации уже существующей системы при переходе на новые технологические решения. В этом случае изменению подвергается только реализация операций, но никак не функциональность самой системы. А это обеспечивает совместимость выпускаемых версий программ при итеративной технологии разработки программных систем.

4.5. Шаблон

Концепция шаблонов является достаточно мощным средством в ООП, и поэтому ее использование в языке UML 2.0 позволяет не только сократить размеры диаграмм моделей программных систем, но и наиболее корректно управлять наследованием свойств и поведения отдельных элементов модели.

Шаблон (template) — классификатор, который в своем описании имеет несколько формальных параметров.

Шаблон является параметризуемым элементом, который может быть использован для получения других элементов модели с помощью отношения связывания шаблона. Параметры шаблона специфицируют формальные параметры, которые должны быть заменены действительными параметрами (или по умолчанию) при связывании. Параметр шаблона определяется в пространстве имен данного шаблона и представляет собой элемент модели, который определяется в контексте этого связывания.

Семантика отношения связывания эквивалентна копированию содержимого шаблона на соответствующий связанный элемент с заменой любых элементов, использованных в качестве параметра шаблона, соответствующим элементом или элементами, специфицированными как действительные параметры в этом связывании. Связанный элемент может иметь несколько связываний, возможно, для одного и того же шаблона. Связанный (не шаблонный) элемент является обычным элементом и, таким образом, может быть использован в качестве несвязанного (и не шаблонного) элемента одного и того же вида. Например, некоторый связанный класс может быть использован в качестве типа типизированного элемента.

Графически шаблон класса изображается прямоугольником класса, на верхний правый угол которого наложен маленький прямоугольник из пунктирных линий (рис. 4.24, *а*). Основной прямоугольник может быть разделен на секции, аналогично обозначению для класса, в верхней секции которого записывается его имя по правилам записи имен для классов. В пунктирном прямоугольнике указывается список формальных параметров шаблона, который не должен быть пустой, хотя он может быть скрыт в данном представлении. Список формальных параметров шаблона может быть изображен в форме списка, разделенного запятыми, или представлен по одному формальному параметру шаблона на линии. Общая нотация для формального параметра шаблона — строка, изображаемая внутри списка параметров шаблона:

<*параметр-шаблона*>::= <*имя-параметра-шаблона*> [':'<*вид-параметра*>] ['='<*по умолчанию*>]

где <*вид-параметра*> является именем метакласса для используемого элемента. Синтаксис <*имя-параметра-шаблона*> зависит от вида параметризуемого элемента для данной подстановки параметра шаблона, а <*по умолчанию*> зависит от вида элемента.

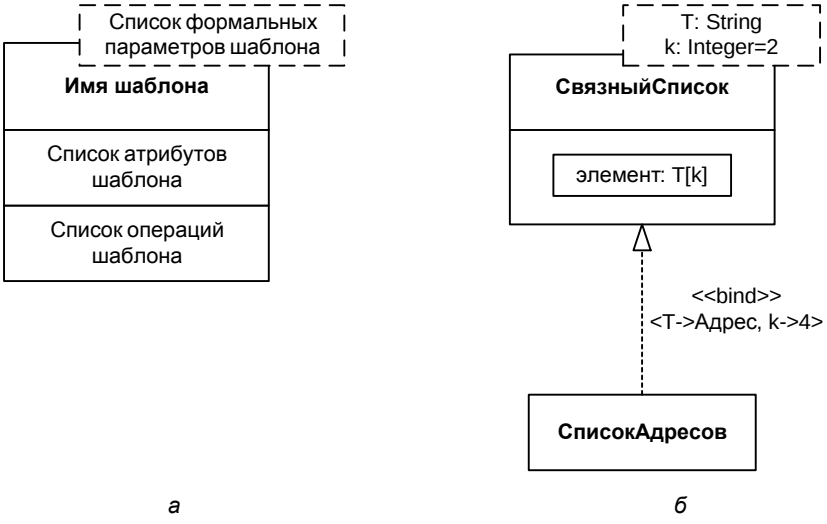


Рис. 4.24. Графическое изображение шаблона на диаграмме классов: *а* — общее представление шаблона; *б* — связывание шаблона

Связывание шаблона является направленным отношением от связанного элемента целевому шаблону (рис. 4.24, *б*). Связывание шаблона владеет множеством подстановок параметров шаблона. Связывание шаблона изображается в форме пунктирной линии с концом на связанном элементе и стрелкой

на шаблоне и с ключевым словом <<bind>>. Эта взаимосвязь означает, что класс-клиент использует некоторый шаблон для своей последующей параметризации. Подстановка параметра шаблона ассоциирует один или более действительных параметров с формальным параметром шаблона. При этом действительный параметр должен быть совместимым с формальным параметром шаблона.

Информация связывания в общем случае изображается в форме разделенного запятыми списка подстановок параметров шаблона:

<*подстановка-параметра-шаблона*>::= <*имя-параметра-шаблона*>'->' <*действительный-параметр-шаблона*>

где синтаксис <*имя-параметра-шаблона*> зависит от вида формального параметра шаблона, а <*действительный-параметр-шаблона*> зависит от вида элемента, используемого в качестве действительного параметра.

Изображенный выше пример (см. рис. 4.24, *б*) отражает тот факт, что связанный класс `СписокАдресов` может быть получен из шаблона `СвязныйСписок` на основе замены формальных параметров `T` и `k` действительными параметрами `Адрес` и `4`. Первый формальный параметр шаблона с именем `T` является строковым параметром. Второй формальный параметр шаблона с именем `k` является целочисленным параметром шаблона, который, хотя и имеет по умолчанию значение `2`, в контексте данного связывания принимает значение `4`.

4.6. Диаграмма классов для системы продажи товаров в интернет-магазине

В качестве примера рассмотрим диаграмму классов для системы продажи товаров в интернет-магазине. Вообще говоря, подобная диаграмма классов может быть построена и представлена различным образом, что отражает субъективный характер разработчика на отдельные аспекты реализации модели. Один из возможных вариантов диаграммы классов для данного примера изображен на рис. 4.25.

Данная диаграмма содержит 5 классов, при этом класс `Покупатель` является элементом этой диаграммы, хотя он был изображен как актер на диаграмме вариантов использования. Что касается состава операций классов и их видимостей, то их выбор определяется, исходя из возможной программной реализации данной модели.

Изображенная диаграмма классов является предварительной статической моделью системы продажи товаров в интернет-магазине и может уточняться на последующих этапах работы над проектом. В частности, может потребоваться

изменить состав операций отдельных классов или их видимость, ввести дополнительные атрибуты или более детально расписать сигнатуру операций. Все эти действия предлагается выполнить читателям самостоятельно в качестве упражнения.

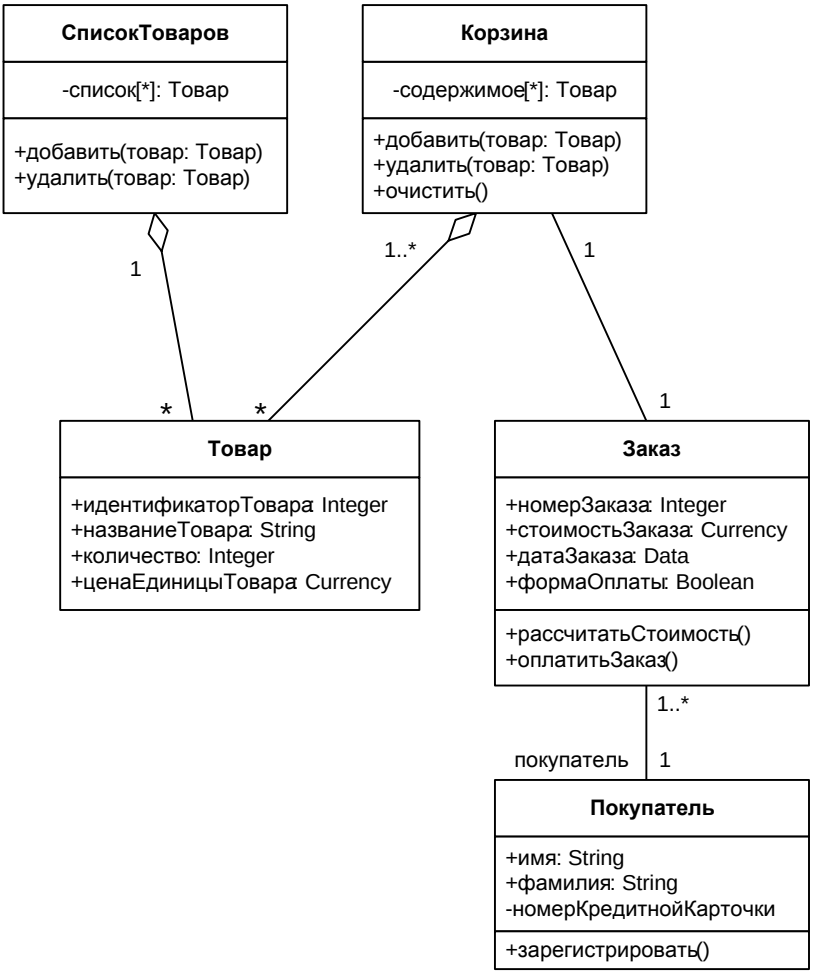


Рис. 4.25. Диаграмма классов системы продажи товаров в интернет-магазине

Общие рекомендации по разработке диаграмм классов, а также перечень типичных ошибок, которые встречаются при их разработке, хотя и не относятся непосредственно к семантике языка UML 2.0, могут оказать существенную помощь разработчикам. При необходимости с этим материалом можно познакомиться по ранее изданным книгам автора, перечень которых приводится в конце книги.

* * *

В заключение представлена сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм классов (табл. 4.4). Хотя в дополнительных профилях к языку UML 2.0 имеются некоторые другие элементы, приводимого перечня может оказаться вполне достаточно для выполнения большинства реальных проектов.

Таблица 4.4. *Графические узлы и пути на диаграммах классов*

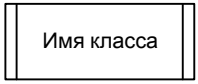
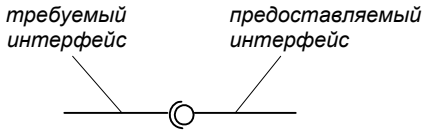
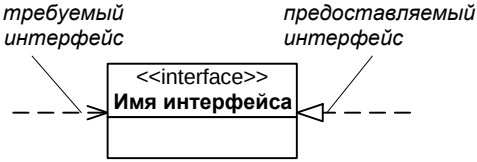
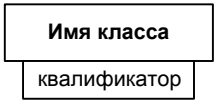
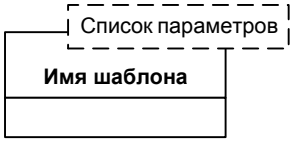
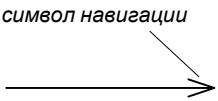
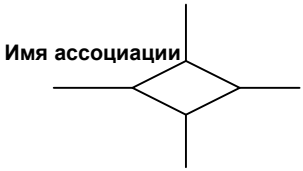
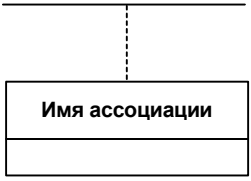
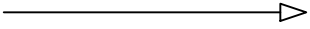
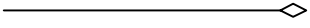
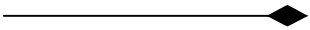
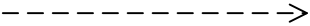
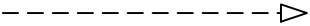
Тип графического элемента	**Нотация**	**Ссылка на раздел**
Класс (class) — символ классификатора		Класс *(см. разд. 4.2)*
Активный класс (active class)		Класс *(см. разд. 4.2)*
Интерфейс (interface) — специальный символ		Интерфейс *(см. разд. 4.4)*
Интерфейс (interface) — символ классификатора		Интерфейс *(см. разд. 4.4)*
Квалификатор (qualifier)		Квалификатор *(см. разд. 4.3.4)*
Шаблон (template)		Шаблон *(см. разд. 4.5)*

Таблица 4.4 *(окончание)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Бинарная ассоциация (binary association)		Ассоциация *(см. разд. 4.3.1)*
n-арная ассоциация (n-ary association)		*N*-арная ассоциация *(см. разд. 4.3.2)*
Ассоциация-класс (association class)		Ассоциация-класс *(см. разд. 4.3.3)*
Обобщение (generalization)		Обобщение *(см. разд. 4.3.5)*
Агрегация (aggregation)		Агрегация *(см. разд. 4.3.7)*
Композиция (composition)		Композиция *(см. разд. 4.3.8)*
Зависимость (dependency)		Зависимость *(см. разд. 4.3.9)*
Реализация (realization)		Реализация *(см. разд. 4.3.10)*

Глава 5

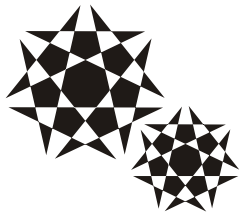


Диаграмма композитной структуры (composite structure diagram)

Диаграмма классов представляет собой логическую модель системы, на которой могут быть изображены взаимосвязи структурного характера только между классами. При этом внутренняя структура классов описывается в виде атрибутов и операций. Этого оказывается недостаточно при моделировании программных систем, имеющих сложную структуру своих составных частей. Именно для этой цели в языке UML 2.0 служит диаграмма композитной структуры, которая позволяет изображать внутреннюю структуру или состав классов.

5.1. Композитная структура

Термин "структура" в этой главе относится к композиции взаимодействующих элементов, представляющих экземпляры времени выполнения, которые кооперируются с помощью связей коммуникации для достижения некоторых общих целей. Структура показывает конфигурацию частей, которые совместно реализуют поведение содержащего их классификатора.

Внутренняя структура (internal structure) — структура взаимодействующих элементов модели, которые создаются в экземпляре содержащего их классификатора.

На диаграммах композитной структуры класс наделяется способностью иметь внутреннюю структуру и порты. Класс выступает в качестве пространства имен для различных видов классификаторов, определенных в его области действия, включающей другие классы. Вложение классов друг в друга ограничивает их видимость внутренней областью пространства имен содержащего

их класса и применяется для целей сокрытия информации. Другие вложенные классификаторы используются подобно любому другому классификатору в содержащем его классе.

Для обеспечения необходимого поведения системы ее элементы обычно кооперируются друг с другом. При этом поведение является той функциональностью, которой должна обладать разрабатываемая система на основе предъявляемых к ней требований. На диаграммах композитной структуры для описания множества экземпляров используются кооперации. Дополнительно указываются специальные роли, которые могут играть соответствующие экземпляры. Роли в кооперации часто могут быть типизированы интерфейсами и, тем самым, предопределять свойства, которые кооперирующие экземпляры должны проявлять извне.

Таким образом, поведение кооперации может быть представлено множеством кооперирующих экземпляров классов, которые взаимодействуют друг с другом посредством передачи сигналов или вызова операций. Однако при моделировании программных систем важно описать только те аспекты взаимодействия элементов кооперации, которые относятся к выполнению некоторой задачи или реализации вариантов использования проектируемой системы.

В общем случае для описания внутренней структуры классификатора используется понятие свойства.

Свойство (property) — множество экземпляров, которые являются собственностью содержащего их экземпляра классификатора.

Когда создается экземпляр классификатора, имеющего некоторую внутреннюю структуру, то множество экземпляров, соответствующих его свойствам, может быть создано либо немедленно, либо спустя некоторое время. Все эти экземпляры являются экземплярами тех классификаторов, которые образуют внутреннюю структуру исходного классификатора. При разрушении экземпляра исходного классификатора все такие экземпляры также разрушаются.

Для представления внутренней структуры классификаторов в общем случае предназначена диаграмма композитной структуры. При этом классификатор, имеющий некоторую внутреннюю структуру, называют также *композитным*, а его внутреннюю структуру — композитной структурой.

5.2. Композитный класс

Как уже отмечалось ранее, композитный класс имеет некоторую внутреннюю структуру, состоящую из элементов, которые представляют собой части или свойства этого класса.

5.2.1. Часть

Часть (part) — свойство, которое является элементом внутренней структуры композитного классификатора, в частном случае — класса.

На диаграммах композитной структуры часть изображается с помощью вложения графического символа прямоугольника с непрерывным контуром, который представляет часть, в символ содержащего ее класса в отдельной его секции. Так, например, класс `Автомобиль` содержит в качестве своей части — свойство `Кузов` (рис. 5.1, *а*). Свойство, на которое может ссылаться экземпляр содержащего его класса, изображается символом прямоугольника с пунктирным контуром, вложенным в содержащий его класс. Так, например, класс `Автомобиль` ссылается на один или два экземпляра класса `Двигатель` (рис. 5.1, *б*).

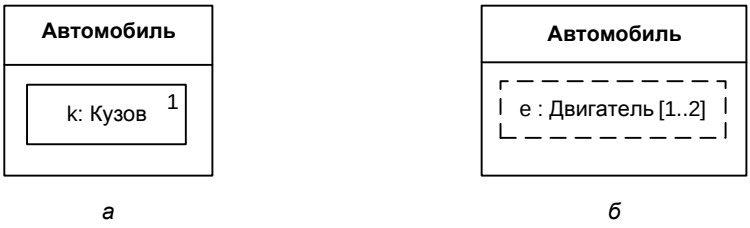


Рис. 5.1. Варианты графического изображения части на диаграмме композитной структуры: *а* — класс содержит свойство; *б* — класс ссылается на экземпляр другого класса

Символ прямоугольника части или свойства имеет только секцию имени, содержащую строку текста, которая должна удовлетворять следующему синтаксису в БНФ:

```
<имя-части>::= [<собственное-имя-части>] [: <имя-класса>]
[[<кратность>]] | [<имя-класса>]
```

где <*собственное-имя-части*> является именем экземпляра класса и обычно записывается со строчной буквы, а <*имя-класса*> является именем соответствующего класса, от которого инстанцируется данная часть или свойство. Кратность части может быть также показана в правом верхнем углу прямоугольника части.

Символ части может содержать имя без двоеточия в строке имени. Это означает анонимность класса, вложенного в пространство имен содержащего его класса. Часть имеет этот анонимный класс в качестве своего типа. Каждый вариант анонимного класса отличается от любого другого его варианта. Разрешается показывать секции атрибутов и операций анонимно именованного класса.

Следует заметить, что, хотя синтаксис позволяет это, <*имя-части*> не может быть пустым.

Поскольку на диаграмме классов может быть изображено отношение композиции, существует определенная взаимосвязь между представлением композитной структуры класса в форме этого отношения и в форме части класса. Так, например, класс `Автомобиль` может иметь два возможных представления. На диаграмме классов класс `Автомобиль` может быть изображен в форме отношения композиции с именем роли `задняяЧасть` для класса `Колесо` и ассоциации с именем роли `e` для класса `Двигатель` (рис. 5.2, *а*). На диаграмме композитной структуры этот же класс `Автомобиль` может быть изображен в форме своих частей (рис. 5.2, *б*).

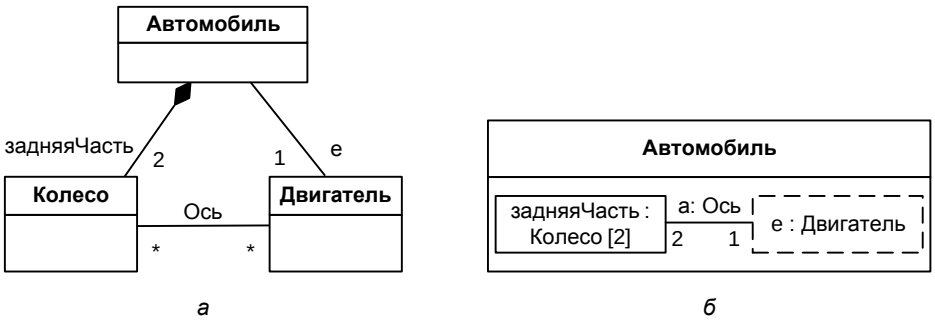


Рис. 5.2. Варианты графического изображения композитного класса: *а* — в форме отношения композиции; *б* — в форме своих частей

Дополнительно фрагмент на рис. 5.2, *б* специфицирует, что `задняяЧасть` принадлежат внутренней структуре класса `Автомобиль`. Это позволяет считать данную спецификацию более детальной, поскольку она поддерживает экземпляры классов `Колесо` и `Двигатель` только в контексте класса `Автомобиль` и не поддерживает колеса и двигатели в общем случае.

Фрагмент на рис. 5.2, *а* специфицирует тот факт, что любой экземпляр класса `Двигатель` может быть связан с произвольным числом экземпляров клас-

са Колесо. Фрагмент на рис. 5.2, *б* специфицирует, что в контексте класса Автомобиль, экземпляр, играющий роль е, может быть связан только с двумя экземплярами класса Колесо, играющими роль задняяЧасть. Более того, экземпляры, играющие роли е и задняяЧасть, могут быть связаны, только если они рассматриваются в контексте одного и того же экземпляра класса Автомобиль.

Таким образом, фрагмент на рис. 5.2, *б* вводит дополнительные ограничения на экземпляры классов Колесо и Двигатель, когда они играют соответствующие роли в экземпляре класса Автомобиль. Эти ограничения не являются истинными для экземпляров Колесо и Двигатель в общем случае. Другие колеса и двигатели могут быть связаны произвольным образом, как это специфицировано на рис. 5.2, *а*.

5.2.2. Соединитель

Соединитель (connector) — отношение, которое обеспечивает взаимосвязь или коммуникацию между двумя или более экземплярами классификаторов, в частном случае — экземплярами классов.

На диаграмме композитной структуры соединитель представляет собой *связь* (link), которая является экземпляром ассоциации. В общем случае соединитель используется для предоставления потенциальной возможности экземплярам взаимодействовать между собой на различных диаграммах. В простых случаях связь может реализовываться как указатель, в более сложных случаях — как сетевое соединение. В противоположность ассоциации, которая специфицирует связи между любыми экземплярами ассоциированных классов, соединители специфицируют связи между экземплярами, являющимися только структурно связанными частями некоторого класса.

Соединитель может быть присоединен к двум или более связанным элементам, каждый из которых представляет множество экземпляров. Каждый конец соединителя является различным в том смысле, что он играет различную роль в коммуникации, реализуемой посредством соединителя. Коммуникации, реализуемые посредством соединителя, могут иметь дополнительные ограничения, которые применяются к присоединенным связанным элементам. Типы связанных элементов, к которым присоединены концы соединителя, должны соответствовать типам концов ассоциации, которая типизирует этот соединитель, если таковая имеется.

Соединитель изображается с использованием нотации для ассоциации. Необязательная строка имени соединителя должна удовлетворять следующему синтаксису:

<*имя-соединителя*>::= ([*имя*] ':' <*имя-класса*>) | <*имя*>

где <*имя*> является собственным именем соединителя, а <*имя-ассоциации*> является именем ассоциации или ее типом. Дополнительно выше или перед именем соединителя может быть помещен стереотип в виде ключевого слова в угловых кавычках. После или ниже имени соединителя может быть помещена строка-свойство.

Другие обозначения на концах соединителя могут специфицировать дополнительные свойства концов предполагаемой ассоциации. Если <*имя-ассоциации*> отсутствует, то тип соединителя является анонимно именованной ассоциацией, концы которой должны соответствовать каждому концу соединителя. Любая ассоциация, соответствующая соединителю, всегда является направленной в обоих направлениях и является собственностью содержащего ее композитного класса.

Связи, соответствующие соединителям, могут быть сформированы при создании экземпляра содержащего их класса. Множество связей является подмножеством полного множества связей, специфицированных ассоциацией, которая соответствует этому соединителю. Все связи разрушаются, когда разрушается содержащий их экземпляр класса.

Конец соединителя (connector end) является концевой точкой соединителя, которой соединитель присоединяется к соединяемому элементу.

Конец соединителя описывает элемент, который присоединен к соединителю и которым владеет этот соединитель. На конце соединителя могут быть изображены дополнительные обозначения, соответствующие дополнительным обозначениям на конце ассоциации. Эти дополнительные обозначения специфицируют свойства ассоциации, которая является типом данного соединителя. При этом тип каждого конца ассоциации должен соответствовать типу части, к которой присоединен соответствующий конец соединителя.

Кратность указывает на количество экземпляров, которые могут быть связаны с каждой ролью в спецификации экземпляра класса на другом конце соединителя. Если кратность не специфицируется, то кратность совпадает с кратностью роли, к которой присоединен этот конец. Каждый конец соединителя является частью только одного соединителя. Кратности на концах соединителя служат для ограничения количества создаваемых начальных связей. Связи должны быть созданы для каждого экземпляра, который играет некоторую роль в соединении до тех пор, пока не будет достигнута минимальная кратность конца соединителя для обоих концов этого соединителя.

В общем случае кратности на структурных характеристиках и концах соединителя указывают количество экземпляров (объектов и связей), которые могут быть созданы в экземпляре содержащего их класса. Нижняя граница интервала кратности указывает на количество экземпляров, которые обязательно создаются, верхняя граница интервала кратности указывает на максимальное количество экземпляров, которые могут быть созданы. Все экземпляры, соответствующие частям структурированного класса, разрушаются рекурсивно, когда разрушается экземпляр содержащего их класса. Кратности соединяемых элементов должны быть согласованными.

5.2.3. Роль в спецификации экземпляра класса

На диаграммах композитной структуры могут быть также изображены спецификации экземпляров класса. При этом для них должны быть указаны имена ролей, которые они играют в композитном классе. Строка имени роли в спецификации экземпляра должна удовлетворять следующему синтаксису в БНФ:

```
<имя-роли-в-спецификации-экземпляра>::= {<имя> ['/'<имя-роли>] |
'/'<имя-роли>} [':' <имя-классификатора> [',' <имя-классификатора>]*]
```

Имя спецификации экземпляра может следовать после имени части, роль которой играет данный экземпляр. Имя части может существовать, только если экземпляр играет эту роль. Применительно к экземплярам класса вся строка имени роли подчеркивается.

Класс `Автомобиль` может служить примером композитного класса, который, например, имеет 4 части, представляющие 4 колеса `Автомобиля`. Все эти части имеют тип `Колесо`, как это изображено на соответствующей диаграмме композитной структуры для класса `Автомобиль` (рис. 5.3). При этом передние и задние колеса соединяются посредством соединителя, представляющего переднюю и заднюю оси соответственно. Данная диаграмма специфицирует тот факт, что всякий раз, когда создается экземпляр класса `Автомобиль`, создаются 4 экземпляра класса `Колесо` и хранятся в форме композиции в экземпляре класса `Автомобиль`. Дополнительно в каждом из экземпляров класса `Автомобиль` создается по одной связи между экземплярами переднее колесо и экземплярами заднее колесо.

Композитный класс `Автомобиль` может быть представлен на диаграмме композитной структуры с помощью указания кратностей своих частей, чтобы показать многократное создание экземпляров класса `Колесо`. В этом случае диаграмма, изображенная на рис. 5.4, специфицирует, что два экземпляра левого колеса и два экземпляра правого колеса в каждом экземпляре класса `Автомобиль` должны быть соединены связью, получаемой от соединителя,

представляющего эту ось. При этом кратности указывают на то, что ни к левой, ни к правой стороне экземпляра класса `Автомобиль` не могут быть добавлены дополнительные экземпляры класса `Колесо`.

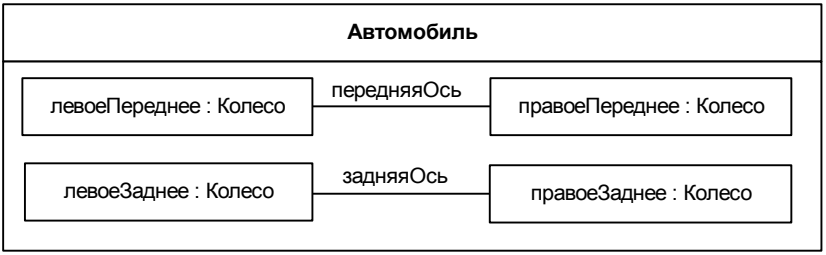


Рис. 5.3. Фрагмент диаграммы композитной структуры для класса `Автомобиль`

Рис. 5.4. Диаграмма композитной структуры для класса `Автомобиль` с использованием кратностей частей

5.3. Порт класса

Язык UML 2.0 предоставляет механизмы для изоляции любого классификатора, и в частности — класса, от его окружения. Это достигается спецификацией некоторой точки взаимодействия между внутренними элементами классификатора и его окружением. Такая точка взаимодействия получила название порта.

Порт (port) — свойство классификатора, которое специфицирует отдельную точку взаимодействия между этим классификатором и его окружением или между классификатором и его внутренними частями.

В общем случае порт предназначен для спецификации сервисов, которые класс предоставляет или предлагает своему окружению, а также сервисов, которые класс ожидает или требует от своего окружения. Если все взаимодействия класса с его окружением осуществляются через его порты, то внут-

ренность класса может быть полностью изолирована от окружения. Это позволяет рассматривать такой класс независимо и повторно использовать его в любом контексте, который удовлетворяет ограничениям, специфицированным с помощью этих портов.

Если запросы, достигающие порта класса, передаются поведению этого класса, то такой порт называется *портом поведения* (behavior port). В этом случае любой вызов характеристики поведения, поступающий в порт поведения, должен быть обработан экземпляром композитного класса, который владеет этим портом, а не экземплярами частей, которые может содержать этот класс. Другими словами, порт поведения специфицирует тот факт, что любые запросы, достигающие этого порта, должны быть обработаны экземплярами владеющего им класса, а не быть переданными дальше любым вложенным экземплярам, если таковые имеются. Однако если для класса никакое поведение не определено, то любые запросы, достигающие порта поведения этого класса, игнорируются или теряются.

Порт класса изображается в форме символа небольшого квадрата, рядом с которым размещается имя порта. Если порт изображен с перекрытием границы символа прямоугольника класса, то это означает, что соответствующий порт открыт для наблюдения извне, т. е. его видимость является общедоступной (рис. 5.5, *а*). Если порт изображен внутри символа прямоугольника класса, то это означает, что этот порт является скрытым, а его видимость по умолчанию является защищенной (рис. 5.5, *б*). Порт поведения изображается в форме символа порта, связанного линией с символом состояния, изображенным внутри символа класса (рис. 5.5, *в*). При этом символ состояния изображает инкапсулированное поведение этого класса.

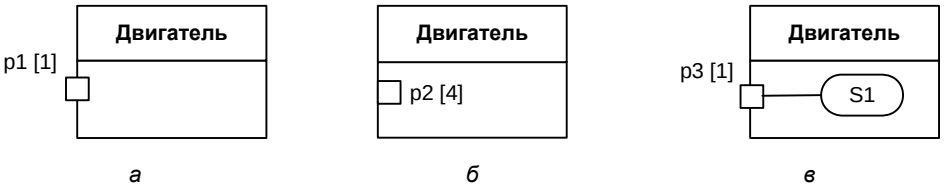


Рис. 5.5. Нотация портов на диаграмме композитной структуры: *а* — класс открыт для наблюдения извне; *б* — скрытый порт; *в* — порт поведения

Рядом с портом может быть изображено его имя и тип, которые разделяются двоеточием. После имени порта может быть указана его кратность, заключенная в прямые скобки. Имя порта, его тип и кратность могут быть опущены. Тем не менее каждый безымянный порт отличается от любого другого порта на диаграмме.

С портами класса могут быть ассоциированы интерфейсы для спецификации характера взаимодействия, которое может осуществляться через этот порт. В этом случае все взаимодействия через порт специфицируются в форме и предоставляемых и требуемых интерфейсов.

Предоставляемый интерфейс (provided interface) порта характеризует запросы, которые могут быть переданы через этот порт классу от его окружения. Предоставляемые интерфейсы порта предназначены для спецификации характеристик поведения или сервисов, которые владеющий класс предлагает своему окружению в этой точке взаимодействия. В частном случае, предоставляемые интерфейсы порта могут специфицировать множество операций, которые класс предлагает своему окружению через данный порт. Запросы, поступающие через предоставляемые интерфейсы порта, должны быть обработаны или непосредственно экземпляром композитного класса, или посредством их пересылки частям его внутренней структуры.

Требуемый интерфейс (required interface) порта характеризует запросы, которые могут быть переданы от класса к его окружению через этот порт. Требуемые интерфейсы порта предназначены для спецификации сервисов, которые класс ожидает от своего окружения и к которым он может иметь доступ через данный порт. В частном случае, требуемые интерфейсы порта могут специфицировать множество операций, выполнения которых класс ожидает от своего окружения. При этом экземпляры этого класса ожидают, что характеристики, которые находятся в собственности требуемых ему интерфейсов, будут предложены одним или более экземпляров других классов из его окружения.

Предоставляемый интерфейс изображается с использованием нотации "леденец на палочке" (рис. 5.6, *а*). Данный фрагмент специфицирует предоставляемый интерфейс `ITрансмиссия` для порта `p1` с кратностью `1` для класса `Двигатель`. Требуемый интерфейс изображается в форме нотации "шарнира" (рис. 5.6, *б*). Данный фрагмент специфицирует требуемый интерфейс `ITопливо` для порта `p2` с кратностью `1` для класса `Двигатель`.

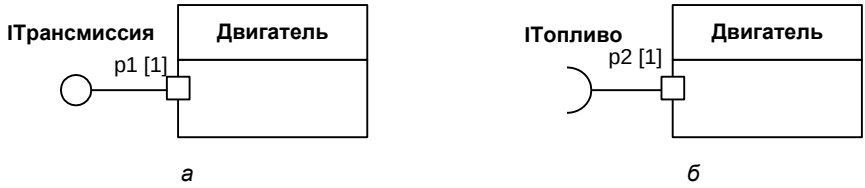


Рис. 5.6. Нотация предоставляемого (*а*) и требуемого (*б*) интерфейсов для порта класса

С целью спецификации различных типов взаимодействий для класса может быть определено несколько портов. При этом отдельные порты могут быть соединены одновременно с предоставляемыми и требуемыми интерфейсами.

Предоставляемые и требуемые интерфейсы полностью характеризуют любое взаимодействие, которое может происходить между классом и его окружением в порту, относительно данных, поступающих в этот порт, и поведения, которое может быть вызвано через этот порт.

Если порт на границе класса связан соединителем с некоторой частью внутренней структуры класса, то это означает, что экземпляр композитного класса должен направлять любые запросы, достигающие этого порта, далее вдоль связи, специфицированной этим соединителем. Если же соединитель присоединен только к одной стороне порта, то любые запросы, достигающие этого порта, будут заканчиваться в этом порту.

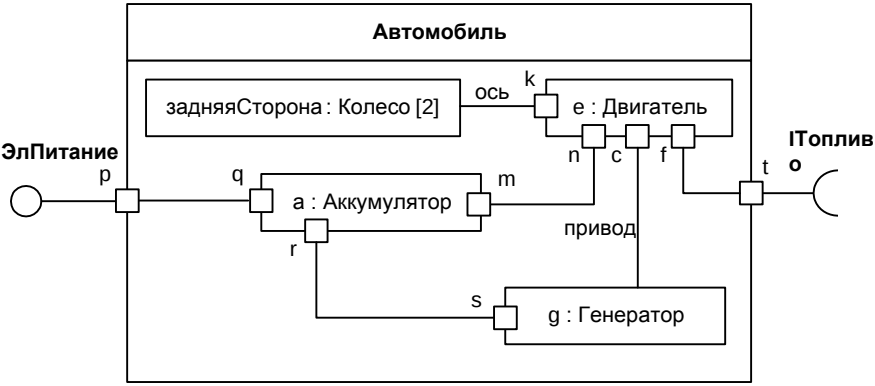


Рис. 5.7. Пример композитной структуры класса `Автомобиль`

Для иллюстрации рассмотрим композитный класс `Автомобиль` (рис. 5.7). В качестве своих составных частей, кроме всего прочего, он имеет `Двигатель`, `Аккумулятор`, `Генератор` и `Колеса`. При этом `Двигатель` через порт `k` связан с двумя ведущими задними колесами посредством оси, через порт `c` связан с `Генератором` посредством `привода`, через порт `n` связан с `Аккумулятором` и через порт `f` связан с портом `t` класса `Автомобиль`. В свою очередь, порт `t` класса `Автомобиль` связан с требуемым интерфейсом `ITопливо`, что характеризует необходимость использования топлива для работы двигателя. При этом все запросы к этому интерфейсу будут исходить не от экземпляров класса `Автомобиль`, а непосредственно от экземпляров класса `Двигатель`. Другими словами, порт `f` специфицирует сервис, который двигатель ожидает

получить от своего окружения, или операции, вызовы которых по коммуникациям могут быть переданы из этого порта.

Класс `Автомобиль` имеет предоставляемый интерфейс `IЭлПитание`, который через порт `p` связан с портом `q` `Аккумулятора`, являющимся внутренней частью класса `Автомобиль`. На практике это характеризует возможность "дать прикурить" другому автомобилю. Аккумулятор через порт `r` связан с портом `s` `Генератора`, что позволяет подзаряжать его во время работы двигателя. Генератор для своей работы должен использовать энергию двигателя, что специфицировано с помощью связи `привод`.

В данном примере класс `Двигатель` является полностью инкапсулированным, т. е. он может быть специфицирован без знания окружения, в которое двигатель будет встроен. До тех пор пока окружение двигателя удовлетворяет ограничениям, выраженным с помощью его портов, любой содержащий этот двигатель автомобиль будет функционировать должным образом.

Таким образом, порты, предназначенные для отделения внутренних элементов класса от его окружения, позволяют композитному классу быть определенным независимо от своего окружения. Это дает возможность повторно использовать данный класс в любом окружении, которое согласуется с ограничениями на взаимодействия, налагаемые его портами.

5.4. Кооперация

Понятие кооперации является одним из фундаментальных понятий в языке UML 2.0. В общем случае кооперация используется для объяснения, как работает система или некоторая ее подсистема. При этом кооперация позволяет определить поведение системы в терминах взаимодействия участников этой кооперации.

Кооперация (collaboration) — классификатор, предназначенный для описания некоторой структуры элементов или ролей, которые выполняют специализированные функции и совместно производят желаемую функциональность.

Цель кооперации заключается в том, чтобы объяснить, как совокупность кооперирующих элементов достигает выполнения совместной задачи или множества задач. Поэтому она обычно включает в себя только те аспекты, которые являются необходимыми для этого объяснения, и скрывает все остальные. Другие детали, такие как указание точных имен реально взаимодействующих экземпляров классов, не указываются. Вместо этого кооперация

содержит описания ролей, которые должны играть ее участники для получения желаемой функциональности.

Роль кооперации (collaboration role) специфицирует требуемое множество характеристик, которые должен иметь соответствующий участник кооперации.

Роли кооперации определяют возможность использования экземпляров классов для решения задач кооперации. Со своей стороны классы, которые соответствуют ролям кооперации, специфицируют все требуемые свойства этих экземпляров.

Для изображения кооперации служит пиктограмма в форме пунктирного эллипса с двумя секциями, в верхней из них указывается имя кооперации. Внутренняя структура кооперации, включающая в себя роли и соединители, может быть изображена в нижней секции внутри пунктирного эллипса (рис. 5.8). Для изображения отдельной роли кооперации используется пиктограмма прямоугольника класса. Внутри этого прямоугольника записывается имя роли, аналогично имени части композитного класса. При этом в качестве типа роли указывается имя класса, который специфицирует характеристики, необходимые для решения задач кооперации. Отношения между ролями изображаются в форме соединителей, которые специфицируют, какие пути коммуникации должны существовать между соответствующими экземплярами классов.

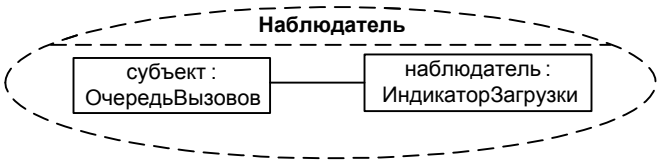


Рис. 5.8. Внутренняя структура кооперации `Наблюдатель`

Поскольку роли кооперации являются ее свойствами, то для представления кооперации может быть использована альтернативная нотация. В этом случае каждое свойство кооперации изображается в форме пиктограммы класса, а именем класса является имя типа соответствующей роли кооперации. Каждый класс соединяется с пиктограммой кооперации линией, которая помечается именем соответствующей роли. Так, например, кооперация `Наблюдатель` может быть представлена в альтернативной форме совместно с реальными классами `ОчередьВызовов` и `ИндикаторЗагрузки`, экземпляры которых участвуют в реализации данной кооперации (рис. 5.9).

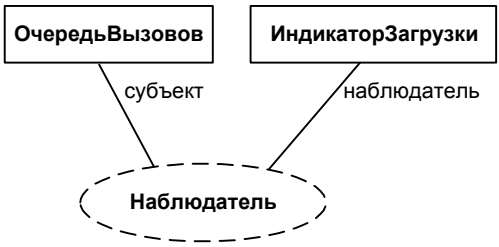


Рис. 5.9. Альтернативное представление кооперации `Наблюдатель`

В этом случае любой экземпляр, играющий роль субъекта, должен обладать свойствами, которые специфицированы в классе `ОчередьВызовов`. Аналогично, любой экземпляр, играющий роль наблюдателя, должен обладать свойствами, которые специфицированы в классе `ИндикаторЗагрузки`.

Поскольку кооперация является разновидностью классификатора, то для представления ее структуры, вообще говоря, может быть использована пиктограмма прямоугольника класса. Однако в этом случае возникает естественная трудность интерпретации ее поведения. С другой стороны, представление кооперации в форме пунктирного эллипса может относиться к отдельному варианту использования и детализировать особенности его последующей реализации, что представляется вполне логичным.

Отдельный экземпляр класса может одновременно играть несколько ролей в одной или нескольких различных кооперациях. Однако каждая кооперация должна специфицировать только те аспекты данного экземпляра класса, которые являются релевантными для цели этой кооперации. Кооперация может быть присоединена к классу или к отдельной операции класса посредством применения кооперации, которая рассматривается далее.

5.5. Применение кооперации

Применение или экземпляр кооперации представляет собой реализацию кооперации в некоторой специальной ситуации или в контексте решения отдельной задачи.

Применение кооперации (collaboration use) представляет собой описание реализации кооперации в форме множества взаимодействующих элементов посредством связывания этих элементов с ролями данной кооперации.

Применение кооперации может относиться к некоторому классификатору или классу. В этом случае применение кооперации устанавливает отношение между некоторой характеристикой в типе ее кооперации с соответствующим атрибутом или операцией класса, который владеет этим применением кооперации.

Применение кооперации изображается пунктирным эллипсом, содержащим имя экземпляра кооперации, двоеточие и имя типа кооперации. Чтобы показать, что применение кооперации относится к классу, используется пунктирная линия зависимости с открытой стрелкой с дополнительным ключевым словом `<<represents>>` в форме стереотипа. При этом стрелка указывает на класс, который владеет данным применением кооперации (рис. 5.10).

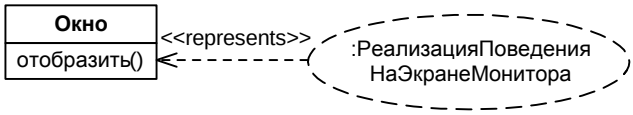


Рис. 5.10. Применение кооперации представляет поведение класса `Окно`

Любое поведение, присоединенное к типу кооперации, применяется к множеству ролей и соединителей, связанных в данном применении кооперации. В этом случае соответствующая кооперация показывает, как экземпляры, соответствующие структурным характеристикам этого класса (например, его атрибуты и части), взаимодействуют для спецификации поведения данного класса. При этом свойства класса отображаются в роли кооперации посредством связывания ролей применения кооперации.

Связывание роли изображается в форме пунктирной линии, соединяющей символ кооперации с элементом модели, как правило, с конкретной частью. Эта пунктирная линия помечается на конце части именем соответствующей роли кооперации. В качестве типа части может использоваться тип соответствующей роли кооперации или имя конкретного класса. При этом один связанный элемент может играть несколько ролей в одном и том же применении кооперации.

В качестве примера рассмотрим кооперацию `Продажа`, которая может быть определена посредством двух частей с именами `продавец` и `покупатель`, представляющих роли в этой кооперации (рис. 5.11, *а*). Эти роли имеют одинаковый тип `УчастникСделки`, который, в свою очередь, связан отношением обобщения с двумя классами: `Компания` и `ФизическоеЛицо` (рис. 5.11, *б*). Эти классы специализируют класс `УчастникСделки`, что позволяет использовать их вместо типа ролей в применении данной кооперации.

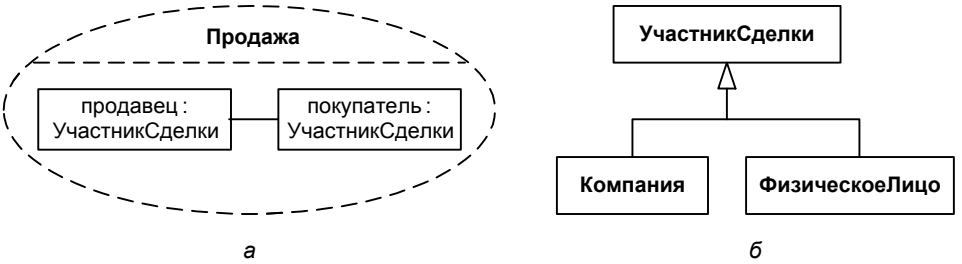


Рис. 5.11. Кооперация `Продажа` (*а*) и специализация класса `УчастникСделки` (*б*)

На основании кооперации `Продажа` могут быть определены два применения этой кооперации с именами `оптовая продажа` и `розничная продажа` (рис. 5.12). Применение кооперации, которое соответствует `оптовой продаже` (рис. 5.12, *а*), имеет две части с именами `производитель` и `дистрибьютор` одного типа `Компания`. При этом `производитель` выступает в роли `продавца` кооперации `Продажа`, а `дистрибьютор` — в роли `покупателя` кооперации `Продажа`. Применение кооперации, которое соответствует `розничной продаже` (рис. 5.12, *б*), имеет две части с именами `дистрибьютор` и `потребитель` с типами `Компания` и `Физическое Лицо` соответственно. При этом `дистрибьютор` выступает в роли `продавца` кооперации `Продажа`, а `потребитель` — в роли `покупателя` кооперации `Продажа`.

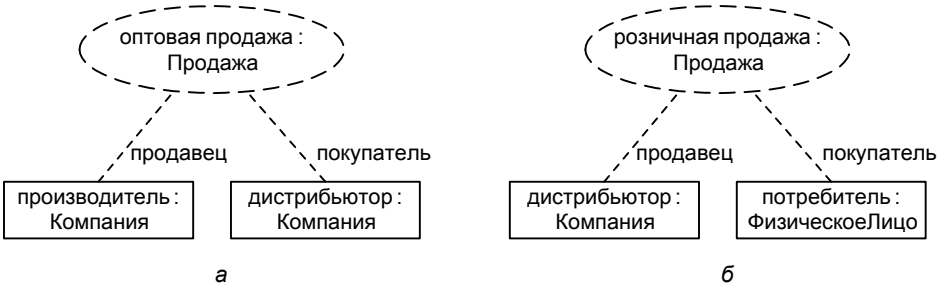


Рис. 5.12. Два применения кооперации `Продажа`:
а — соответствующее `оптовой продаже`;
б — соответствующее `розничной продаже`

В качестве более сложного примера рассмотрим определение кооперации `ПродажаЧерезПосредников`, которая имеет три роли: `производитель`, `посредник` и `потребитель` (рис. 5.13). Эта кооперация содержит два применения кооперации `Продажа` с именами `оптовая продажа` и `розничная продажа`. Первая

из них соответствует кооперации Продажа, в которой производитель является продавцом, а дистрибьютор — покупателем. Вторая из них соответствует кооперации Продажа, в которой дистрибьютор является продавцом, а потребитель — покупателем.

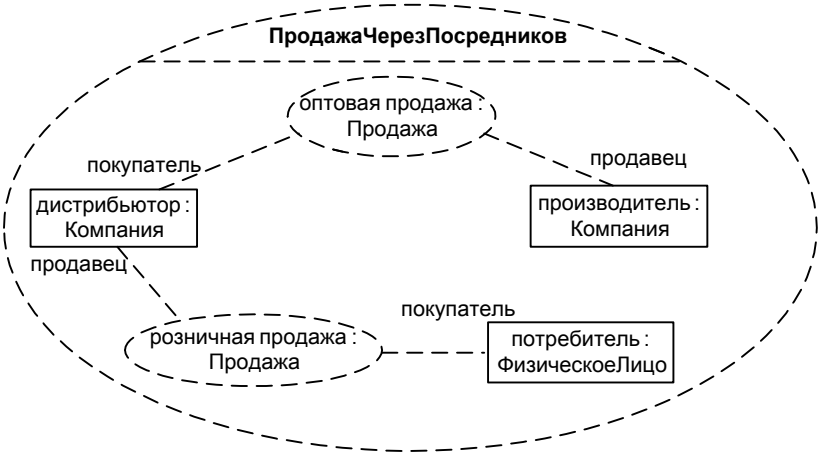


Рис. 5.13. Кооперация Продажа через посредников

В применениях кооперации соединители между продавцами и покупателями могут не показываться. Так в рассматриваемом примере эти соединители являются неявными в кооперации ПродажаЧерезПосредников, поскольку виртуально они входят в состав применяемых кооперации Продажа. Следует заметить, что в кооперации ПродажаЧерезПосредников часть с именем дистрибьютор играет две роли — покупателя и продавца.

Кооперации и их применения могут использоваться для спецификации паттернов проектирования. В этом случае применение кооперации показывает, как паттерн, описываемый кооперацией, применяется в заданном контексте или в некоторой специальной ситуации. Для спецификации паттернов проектирования могут быть также использованы шаблоны кооперации, которые рассматриваются далее в этой главе.

5.6. Шаблон кооперации

Поскольку кооперация является классификатором, то для нее может быть применена концепция шаблона *(см. разд. 4.5)*. В этом случае шаблон кооперации является параметризованной кооперацией, которой соответствует целое семейство кооперации. Параметрами шаблона кооперации являются

типы ее частей или ролей. Отдельная кооперация этого семейства может быть получена связыванием параметров шаблона кооперации с конкретными классами. Шаблоны кооперации находят практическое применение при рассмотрении стандартных паттернов проектирования.

Графически шаблон кооперации изображается пунктирным эллипсом с дополнительным пунктирным прямоугольником в правой верхней своей части (рис. 5.14). Пунктирный эллипс может быть разделен на две секции. В верхней секции записывается имя шаблона кооперации по правилам записи имен для классификаторов. В нижней секции изображаются части или роли шаблона кооперации с соединителями. В пунктирном прямоугольнике указывается список формальных параметров шаблона, который не должен быть пустым, хотя он может быть скрыт в данном представлении. Формальные параметры шаблона кооперации обычно изображаются в виде списка типов, которые разделяются запятыми.

В качестве примера рассмотрим шаблон кооперации с именем `ПаттернНаблюдатель`, который имеет два формальных параметра шаблона: `ТипСубъекта` и `ТипНаблюдателя` (рис. 5.14). Оба формальных параметра шаблона являются типами ролей параметризированной кооперации.

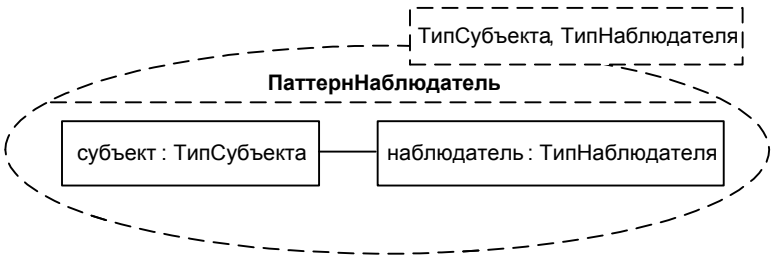


Рис. 5.14. Шаблон кооперации `ПаттернНаблюдатель`

Ранее в качестве примера кооперации была рассмотрена кооперация `Наблюдатель` (см. рис. 5.8 и 5.9). Эта кооперация может быть получена от шаблона кооперации `ПаттернНаблюдатель` (см. рис. 5.14) посредством связывания его параметров `ТипСубъекта` и `ТипНаблюдателя` с классами `ОчередьЗвонков` и `ИндикаторЗагрузки` соответственно (рис. 5.15). При этом отношение связывания изображается в форме отношения зависимости со стереотипом `<<bind>>`.

В качестве альтернативной нотации для связанной кооперации может быть использовано представление информации о связывании в секции внутренней структуры символа кооперации. Так, например, связанная кооперация `Наблюдатель`, полученная от шаблона кооперации `ПаттернНаблюдатель`, может быть изображена так, как показано на рис. 5.16.

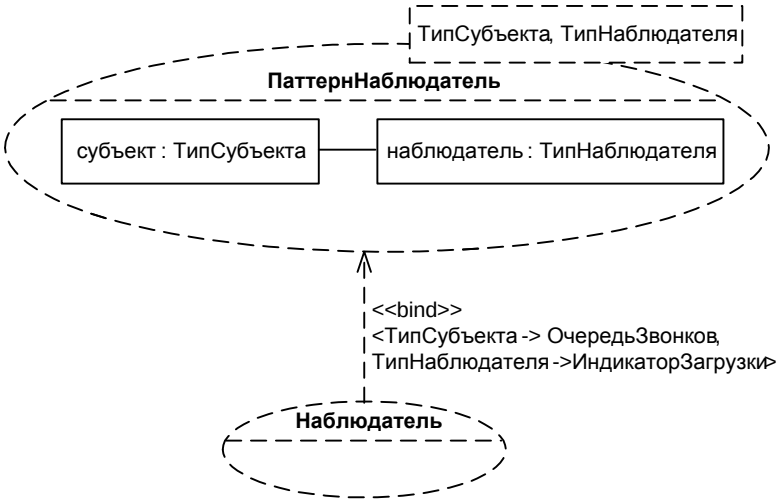


Рис. 5.15. Связывание кооперации `Наблюдатель` с шаблоном кооперации

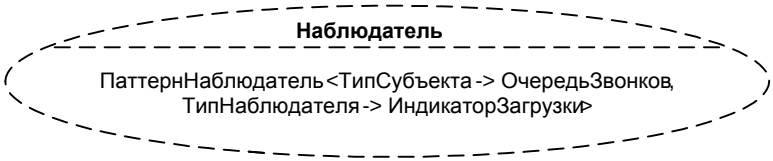


Рис. 5.16. Альтернативное изображение связанной кооперации `Наблюдатель`

Если в структуре шаблона кооперации имеется соединитель, то типом этого соединителя является анонимная ассоциация, концы которой соответствуют именам ролей соответствующих концов соединителя. В этом случае типом каждого конца ассоциации является класс, который реализует часть или роль, связанную соединителем в шаблоне кооперации.

Следует помнить, что связанная кооперация является кооперацией, в то время как применение кооперации является экземпляром кооперации.

К параметрам шаблона могут быть применены ограничения, которые оказывают влияние на действительные классы, делая их ограниченными согласно следующим правилам:

- если параметр шаблона имеет обобщение с некоторым классом, то действительный класс должен иметь обобщение с тем же общим классом;

- если параметр шаблона не имеет обобщения, то действительным классом может быть любой класс;
- если к параметру шаблона не применяются ограничения, то действительным классом может быть любой класс.

В качестве упражнения можно предложить читателям представить кооперацию `Продажа` (см. рис. 5.11) в форме шаблона кооперации, а кооперацию `ПродажаЧерезПосредников` (см. рис. 5.13) изобразить в форме связанной кооперации.

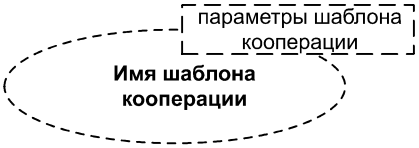
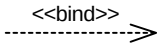
* * *

В заключение приводится сводка основных графических обозначений, которые могут быть использованы при построении диаграмм композитной структуры (табл. 5.1). Все графические узлы и пути, представленные на диаграммах композитной структуры, могут изображаться также на других диаграммах структуры.

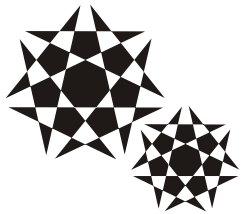
Таблица 5.1. *Графические узлы и пути на диаграммах композитной структуры*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Часть (part)	имя части: Имя класса	Часть *(см. разд. 5.2.1)*
Порт (port)	имя порта: Имя классификатора □	Порт класса *(см. разд. 5.3)*
Кооперация (collaboration**)**	**Имя кооперации**	Кооперация *(см. разд. 5.4)*
Применение кооперации (collaboration use)	имя применения кооперации: Имя кооперации	Применение кооперации *(см. разд. 5.5)*
Соединитель (connector)	———	Соединитель *(см. разд. 5.2.2)*

Таблица 5.1 *(окончание)*

Тип графического элемента	Нотация	Ссылка на раздел
Связывание роли (role binding)		Применение кооперации *(см. разд. 5.5)*
Шаблон кооперации (collaboration template)		Шаблон кооперации *(см. разд. 5.6)*
Связывание параметра шаблона кооперации (parameter binding)		Шаблон кооперации *(см. разд. 5.6)*

Глава 6



Дополнительные диаграммы структуры

В этой главе рассматриваются два типа канонических диаграмм, которые относятся к диаграммам структуры языка UML 2.0: диаграмма пакетов и диаграмма объектов. Хотя эти диаграммы являются дополнительными для представления структуры модели на логическом уровне и содержат небольшое число графических элементов и отношений, их значение весьма различно. Диаграмма пакетов служит основным средством для представления архитектуры программных систем, а также повсеместно используется для спецификации всех основных конструкций метамодели языка UML 2.0. Диаграмма объектов при разработке моделей играет второстепенную роль, поскольку имеющиеся в языке UML 2.0 диаграммы композитной структуры и коммуникации во многом перекрывают графические возможности диаграммы объектов.

6.1. Диаграмма пакетов

Диаграмма пакетов (package diagram) предназначена для представления размещения элементов модели в пакетах и спецификации зависимостей между пакетами и их элементами.

Отдельные фрагменты диаграммы пакетов рассматривались ранее при описании метамодели языка UML 2.0 *(см. разд. 2.3)*. В этой главе представлено систематическое описание всех конструкций, которые могут быть изображены на диаграмме пакетов.

6.1.1. Пакет

Пакет (package) — элемент модели, используемый для группировки других элементов модели.

Элементы модели, которые входят в состав некоторого пакета, называются *членами* этого пакета. Пакет владеет всеми своими членами. В свою очередь, про члены пакета говорят, что они находятся в собственности пакета или принадлежат ему. Если некоторый пакет удаляется из модели, то из модели также удаляются все члены, которые находятся в собственности этого пакета.

Элементы модели, которые входят в пространство имен некоторого пакета, называются *элементами* этого пакета. Среди них могут находиться как члены этого пакета, так и члены других импортируемых пакетов.

На диаграммах пакетов одни пакеты могут быть вложены в другие пакеты. В этом случае вложенный пакет называется *подпакетом*, а все элементы подпакета будут также принадлежать любому пакету, для которого рассматриваемый подпакет является вложенным. Тем самым для элементов модели может быть задано отношение вложенности пакетов, которое представляет собой строгую иерархию.

Как уже было описано ранее, пакет изображается в форме большого прямоугольника, к верхней левой стороне которого присоединен маленький прямоугольник (см. рис. 2.2).

Перед именем пакета может помещаться строка текста, содержащая ключевое слово в форме стереотипа, например: `<<facade>>`, `<<framework>>`, `<<model>>`, `<<stub>>` и `<<topLevel>>`.

Напомним, что содержимое пакетов может быть изображено на диаграммах двумя способами. Первый способ связан с изображением членов пакета в большом прямоугольнике. В этом случае имя пакета размещается в верхней вкладке (см. рис. 2.3). Альтернативно члены пакета могут быть изображены снаружи пакета и соединены с пакетом отрезком линии со знаком плюс (+) в круге на конце, присоединенном к пакету. В этом случае имя пакета размещается в большом прямоугольнике (см. рис. 2.4).

В качестве примера рассмотрим несколько вариантов представления пакета `Простейшие Типы` (см. рис. 2.6). Поскольку на диаграмме пакетов вовсе необязательно изображать пакеты со всем своим содержимым, то этот пакет может быть показан без своих членов (рис. 6.1, *а*). Ранее было описано, что пакет `Простейшие Типы` в языке UML 2.0 включает в себя следующие типы: `Boolean`,

`Integer`, `String` и `UnlimitedNatural` *(см. разд. 2.4.4)*. Это можно визуализировать любым из двух способов: изобразив указанные типы внутри символа пакета (рис. 6.1, *б*) или изобразив типы снаружи символа пакета (рис. 6.1, *в*).

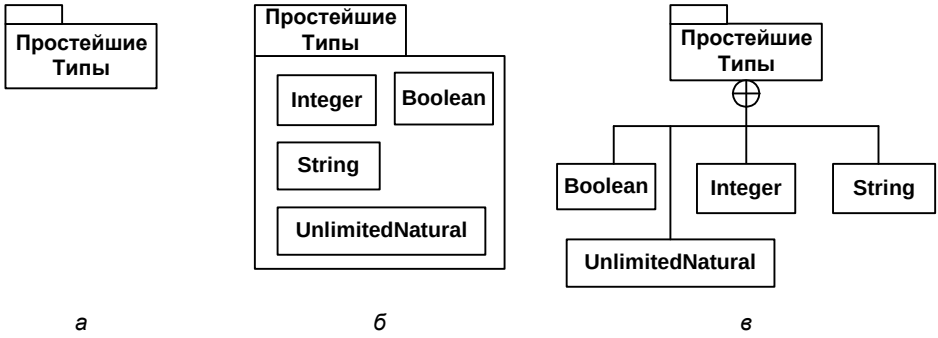


Рис. 6.1. Варианты изображения пакета `Простейшие Типы`: *а* — пакет без своих членов; *б* — тип указан внутри символа пакета; *в* — тип указан снаружи символа пакета

Пакет является пространством имен для всех своих элементов. Это значит, что на элементы пакета можно ссылаться с использованием неквалифицированных имен, т. е. без указания имени содержащего их пакета. Любой элемент, который входит в пакет, может иметь видимость. В общем случае она может быть общедоступной (public) или закрытой (private). Элементы пакета с общедоступной видимостью всегда являются доступным за пределами пакета посредством использования квалифицированных имен, т. е. с указанием имени содержащего их пакета. Видимость элемента пакета может быть указана с помощью символа видимости, предшествующего имени элемента (+ для общедоступного и – для закрытого).

CASE-средства могут изображать видимость элементов пакета с помощью дополнительных специальных символов или выборочно показывать те элементы, которые имеют заданную видимость, например, только общедоступные элементы.

Кроме членов, находящихся в собственности пакета, использовать неквалифицированные имена можно для импортируемых элементов или элементов пакета, в который вложен рассматриваемый пакет. При этом каждый из собственных или импортируемых элементов может иметь свою видимость, которая будет определять доступность этих элементов за пределами содержащего их пакета. Отношение импорта пакетов и элементов рассматривается далее в этой главе.

На диаграмме пакетов допускается изображать не только пакеты, но и отдельные классы, которые, в свою очередь, могут иметь свою собственную структуру. Так, например, диаграмма пакетов `Автомобили` может содержать класс `Автомобиль`, который имеет дополнительную диаграмму композитной структуры (рис. 6.2).

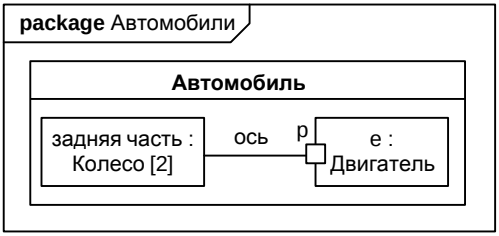


Рис. 6.2. Диаграмма пакетов с классом `Автомобиль`

Наиболее общим отношением между пакетами является отношение зависимости, которое позволяет представить различные по характеру семантические взаимосвязи пакетов на диаграмме.

6.1.2. Зависимость пакетов

Отношение зависимости было рассмотрено ранее *(см. главу 4)* и в общем случае может соответствовать произвольной семантической взаимосвязи между двумя элементами модели или двумя множествами таких элементов. Наиболее часто отношение зависимости используется для спецификации ситуации, когда изменение одного элемента модели может потребовать изменения другого зависящего от него элемента.

Напомним, что отношение зависимости графически изображается пунктирной линией между соответствующими элементами с открытой стрелкой на одном из ее концов. На диаграмме пакетов отношение зависимости может связывать два отдельных пакета, при этом стрелка должна быть всегда направлена от пакета-клиента зависимости (зависимого пакета) к пакету-источнику (независимому пакету).

Так, например, общая модель трехуровневой архитектуры доступа к удаленной базе данных может быть представлена так, как показано на рис. 6.3. На данной диаграмме изображены три пакета: `Уровень Клиента`, `Уровень Бизнес-логики` и `Уровень Данных`, которые дополнительно помечены стереотипом `<<model>>`. При этом пакет `Уровень Данных` является источником зависимости для пакета `Уровень Бизнес-логики`, который, в свою очередь, является источником зависимости для пакета `Уровень Клиента`.

Рис. 6.3. Модель трехуровневой архитектуры доступа к удаленной базе данных

Поскольку пакет является пространством имен, он может импортировать или индивидуальные члены из других пакетов, или все члены других пакетов. Дополнительно содержимое пакета может сливаться с другими пакетами. Эти отношения являются предметом дальнейшего рассмотрения.

6.1.3. Импорт пакета

Импорт пакета (package import) — направленное отношение между пакетами, при котором члены одного пакета могут быть добавлены в пространство имен другого пакета.

Импорт пакета является специализацией отношения зависимости между импортирующим пакетом (клиентом зависимости) и импортируемым пакетом (источником зависимости). Импорт пакета позволяет ссылаться только на общедоступные члены импортируемого пакета в другом пространстве имен, используя при этом неквалифицированные имена. Другими словами, импортирующий пакет добавляет имена всех элементов с общедоступной видимостью из импортируемого пакета к своему пространству имен. Концептуально импорт пакета эквивалентен импорту элемента для каждого индивидуального элемента импортируемого пространства имен, за исключением случая, когда уже существует отдельно определенный импорт элемента.

Импорт пакета изображается с использованием пунктирной линии с открытой стрелкой от импортирующего пространства имен к импортируемому пакету. Рядом с пунктирной линией обязательно изображается одно из двух ключевых слов для указания вида импорта пакета: `<<import>>` — для *общедоступного* импорта пакета, и `<<access>>` — для *закрытого* импорта пакета.

При общедоступном (стереотип `<<import>>`) импорте пакета импортируемые элементы могут быть видимы за пределами импортирующего пакета. Если же импорт пакета является закрытым (стереотип `<<access>>`), то импортируемые элементы не могут быть видимы за пределами импортирующего пакета. По умолчанию импорт пакета является общедоступным.

В качестве примера рассмотрим фрагмент диаграммы пакетов, которая иллюстрирует различные виды импорта пакетов (рис. 6.4). В этом случае общедоступные члены пакета `Простейшие Типы` импортируются в пакет `Корзина`,

после чего они далее, вместе с общедоступными собственными членами пакета `Корзина`, импортируются в пакет `Интернет-магазин`. С другой стороны, общедоступные члены пакета `Вспомогательные Элементы` импортируются только в пакет `Корзина`. Тем самым на них нельзя ссылаться с использованием неквалифицированных имен из пакета `Интернет-магазин`.

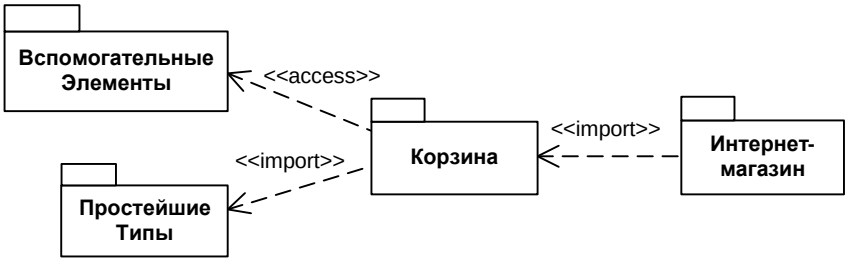


Рис. 6.4. Примеры общедоступного и закрытого импорта пакетов

Кроме импорта всех членов пакета в языке UML 2.0 допускается импортировать и отдельные элементы пакета.

6.1.4. Импорт элемента

Импорт элемента используется для выборочного импорта индивидуальных элементов без импорта всего пакета.

Импорт элемента (element import) — направленное отношение между импортирующим пространством имен и отдельным элементом пакета, которое позволяет ссылаться на этот элемент с использованием неквалифицированного имени.

Импорт элемента является специализацией отношения зависимости между импортирующим пакетом (клиентом зависимости) и импортируемым элементом (источником зависимости). Импорт элемента позволяет идентифицировать отдельный элемент одного пакета в другом пакете посредством добавления имени этого элемента или его псевдонима к пространству имен импортирующего пакета. При этом можно также управлять дальнейшим импортом этого элемента с использованием импорта элемента или импорта пакета.

Импорт элемента можно представить в форме ссылки на импортируемый элемент. Это значит, что в импортирующем пакете невозможно изменить характеристики импортируемого элемента, поскольку модифицировать импортируемый элемент можно только в том пакете, которому он принадлежит или, другими словами, членом которого он является.

Импорт элемента изображается с использованием пунктирной линии с открытой стрелкой, направленной из импортирующего пространства имен к импортируемому элементу. Если видимость является *общедоступной*, то рядом с пунктирной линией изображается стереотип `<<import>>`. Для указания *закрытой* видимости рядом с пунктирной линией изображается стереотип `<<access>>`.

В качестве примера рассмотрим фрагмент диаграммы пакетов, которая иллюстрирует различные виды импорта элементов (рис. 6.5). Импорт элемента для типа данных `Time` (`Время`) из пакета `Типы` является общедоступным, что позволяет элементам в пакете `Программа` ссылаться на тип `Time` без квалификации. Тип `String` (`Строка`) может быть использован в пакете `Программа`, но его нельзя импортировать далее из этого пакета, поскольку импорт этого элемента является закрытым. Однако чтобы в пакете `Программа` ссылаться на тип `Integer` (`Целочисленный тип`), необходимо использовать квалифицированное имя в форме: `Типы::Integer`, поскольку этот элемент в пакет `Программа` не импортируется.

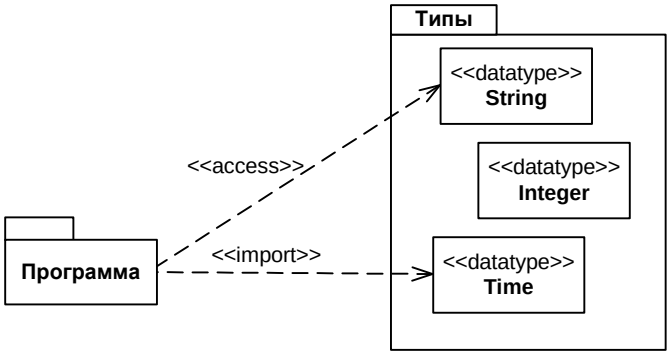


Рис. 6.5. Примеры общедоступного и закрытого импорта элементов

Если неквалифицированное имя импортируемого элемента совпадает с именем любого другого элемента, который находится в импортирующем пространстве имен, то, чтобы его можно было использовать, имя этого импортируемого элемента должно быть квалифицированным. При этом квалифицированное имя элемента не добавляется к импортирующему пространству имен.

Если же имя импортируемого элемента необходимо добавить в пространство имен импортирующего пакета, то вместо имени этого импортируемого элемента необходимо специфицировать *псевдоним* (alias). Такой псевдоним не должен совпадать с именем любого другого члена в импортирующем пакете.

Имя псевдонима может быть изображено после или ниже ключевого слова <<import>> рядом со стрелкой зависимости. По умолчанию имя псевдонима при импорте элемента не используется.

Таким образом, если импорт элемента имеет некоторый псевдоним, то он может использоваться вместо имени этого импортируемого элемента. Так, например, следующий фрагмент изображает импорт элемента с псевдонимом (рис. 6.6). Это означает, что на тип Типы::Real в пакете Формы можно ссылаться как на Double.

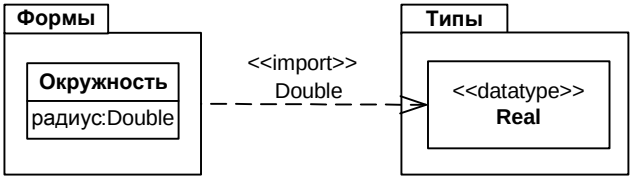


Рис. 6.6. Пример импорта элемента с псевдонимом

По умолчанию предполагается, что видимость импортируемого элемента в импортирующем пакете такая же, как и видимость этого элемента в пакете, из которого он импортируется. Однако эта видимость может быть дополнительно специфицирована. Если видимость импортируемого элемента не специфицирована, то можно добавить видимость к импорту этого элемента. При этом импортируемый элемент имеет или общедоступную видимость, или не имеет видимости вовсе.

Импорт элемента альтернативно может быть представлен в форме текста, который единственным образом идентифицирует импортируемый элемент в фигурных скобках или ниже, или после имени пространства имен. Текстовый синтаксис имеет следующий вид:

```
'{import' <квалифицированное-имя> '}' |
'{access' <квалифицированное-имя> '}'
```

Дополнительно может быть показано имя псевдоним:

```
'{import' <квалифицированное-имя> 'as' <псевдоним> '}' |
'{access' <квалифицированное-имя> 'as' <псевдоним> '}'
```

Таким образом, при использовании в модели импорта пакетов и элементов следует помнить следующее правило. В результате импорта других пакетов или элементов может оказаться, что в пространство имен импортирующего пакета импортируется более чем один элемент с тем же самым именем. Все эти элементы могут быть использованы в импортирующем пакете только

с квалифицированными именами. При этом сами эти элементы в импортирующее пространство имен не добавляются.

CASE-средства могут дополнительно визуализировать видимость элементов, использование которых становится доступным в пакете посредством импорта пакета или импорта элемента. Например, чтобы показать невозможность модификации импортируемых элементов в импортирующем пакете, соответствующие графические символы таких элементов могут быть изображены серым цветом.

6.1.5. Слияние пакетов

Слияние пакетов (package merge) — направленное отношение между двумя пакетами, один из которых расширяет свое содержание посредством добавления содержимого другого пакета.

Слияние пакетов является специализацией отношения зависимости. При этом зависимый пакет получил название *источника* (source), а независимый пакет — *цели* (target). В результате слияния пакетов содержимое пакета-источника расширяется за счет добавления к нему содержимого пакета-цели. Слияние пакетов похоже на отношение обобщения в том смысле, что пакет-источник концептуально добавляет характеристики пакета цели к своим собственным характеристикам, образуя в результате пакет, который объединяет характеристики обоих пакетов.

Таким образом, концептуально слияние пакетов можно рассматривать как операцию, которая на основе содержимого двух пакетов создает новый пакет, объединяющий содержимое используемых при слиянии пакетов. В терминах семантики языка UML 2.0 не существует различия между моделью с явным слиянием пакетов и моделью, в которой все слияния уже выполнены.

Слияние пакетов изображается пунктирной линией с открытой стрелкой, проведенной от пакета-источника к пакету-цели. Дополнительно рядом с этой пунктирной линией изображается стереотип `<<merge>>` (рис. 6.7).

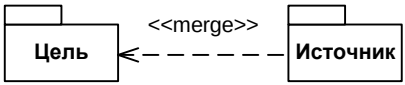


Рис. 6.7. Нотация для отношения слияния пакетов

Слияние пакетов целесообразно использовать, когда элементы, определенные в различных пакетах, имеют одно и то же имя и предназначены для

представления одного и того же понятия. Наиболее часто оно используется для представления нескольких определений одного и того же понятия для различных целей, начиная с некоторого общего базового определения. Данное базовое понятие расширяется посредством приращений, при этом каждое приращение определено в отдельном пакете, который последовательно сливается с другими пакетами. С помощью выбора приращения для слияния можно получить нужное определение некоторого понятия для специальной цели. Слияние пакетов является необходимым механизмом при определении метамодели UML 2.0.

Если элементы в сливаемых пакетах представляют одну и ту же сущность, то характеристики этих элементов концептуально объединяются в отдельном результирующем элементе согласно формальным правилам слияния пакетов, которые рассматриваются далее. При этом слияние пакетов в модели только лишь влечет выполнение этих преобразований, но не результаты преобразований, которые не включаются в модель. В то же время пакет-источник и его содержимое считаются необходимыми для представления результата слияния.

Таким образом, любая ссылка на элемент модели, содержащийся в пакете-источнике, влечет ссылку на результаты слияния. При этом само приращение физически не включается в пакет источник.

Эта особенность отношения слияния может быть проиллюстрирована следующим образом. Рассмотрим фрагмент диаграммы (рис. 6.8), на которой два пакета `P1` и `P2` определяют различные приращения одного и того же класса `A`. Пакет `P2` сливается с содержимым пакета `P1`, что влечет слияние приращения `P1::A` в приращение `P2::A`. Пакет `P3` импортирует содержимое пакета `P2`, поэтому в нем можно определить некоторый подкласс `A`, называемый `SubA`. В этом случае класс `A` в пакете `P3`, т. е. `P3::A`, представляет результат слияния `P1::A` в `P2::A`, а не только лишь приращение `P2::A`. Следует заметить, что если какой-либо другой пакет будет импортировать `P1`, то ссылка на `A` в импортирующем пакете будет представлять приращение `P1::A`, а не результат слияния `P2::A`.

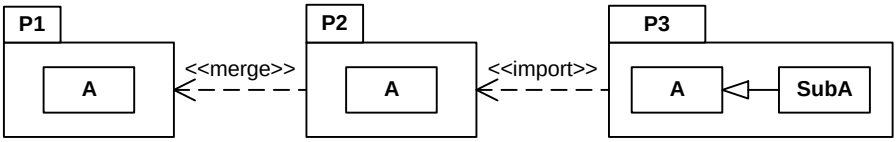


Рис. 6.8. Иллюстрация семантики слияния пакетов

Для понимания правил слияния пакетов необходимо отчетливо различать 3 разные сущности: сливаемое приращение, например, `P1::A` на рис. 6.8,

принимающее приращение, например, `P2::A`, и результат преобразования слияния. Главная трудность проистекает из того факта, что пакет-источник и его содержимое служат как для представления операнда, так и для представления результатов слияния пакетов. При этом конкретная интерпретация зависит от контекста, в котором рассматривается слияние пакетов.

Так, например, пакет `P2`, рассматриваемый в контексте операции слияния пакетов, представляет приращение, которое является операндом этого слияния (рис. 6.8). Однако этот же пакет `P2`, рассматриваемый в контексте импорта пакета, представляет результат слияния пакетов.

Эта двойственная интерпретация одного и того же элемента модели может привести к недоразумениям. Чтобы внести ясность в понимание механизма слияния, дополнительно вводятся в рассмотрение следующие понятия:

- *сливаемый пакет* (merged package) — первый операнд слияния, т. е. пакет, который сливается в принимающий пакет и который является целью стрелки слияния на диаграммах;
- *принимающий пакет* (receiving package) — второй операнд слияния, т. е. пакет, который концептуально содержит результаты слияния и который является источником стрелки слияния на диаграммах. Этот термин используется, чтобы ссылаться на пакет и его содержимое до того, как будут выполнены преобразования слияния;
- *результирующий пакет* (resulting package) — пакет, который концептуально содержит результаты слияния. Этот термин используется, чтобы ссылаться на пакет и его содержимое после того, как слияние будет выполнено. Графически в модели результирующий пакет представляется тем же символом, что и принимающий пакет;
- *сливаемый элемент* (merged element) — элемент модели, который находится в сливаемом пакете;
- *принимающий элемент* (receiving element) — элемент модели, который находится в принимающем пакете. Если этот элемент совпадает со сливаемым элементом, то они оба объединяются для получения результирующего элемента. Этот термин используется, чтобы ссылаться на элемент до выполнения слияния, т. е. на само приращение, а не на его результат;
- *результирующий элемент* (resulting element) — элемент модели в результирующем пакете после выполнения слияния. Для принимающего элемента, который совпадает со сливаемым элементом, это тот же элемент, что принимающий элемент, но в состоянии после того, как было выполнено слияние. Для сливаемых элементов, которые не совпадают с принимающим элементом, он является сливаемым элементом. Для принимающих

элементов, которые не совпадают со сливаемым элементом, он тот же, что и принимающий элемент;

- *тип элемента* (element type) — любой допустимый тип элемента в языке UML 2.0, например, тип параметра или атрибута;
- *метатип элемента* (element metatype) — является типом MOF элемента модели, например, `Классификатор`, `Ассоциация`, `Характеристика`.

Данная терминология предназначена для иллюстрации концептуального представления семантики слияния пакетов посредством следующей схемы (рис. 6.9), которая, строго говоря, не является диаграммой UML. В этом случае собственные элементы пакетов `A` и `B` объединяются в пространство имен пакета `B'`. Однако это представление является всего лишь удобной иллюстрацией описания семантики слияния пакетов и не представляется в модели, поскольку при наличии слияния пакетов реальная модель никак не преобразуется.

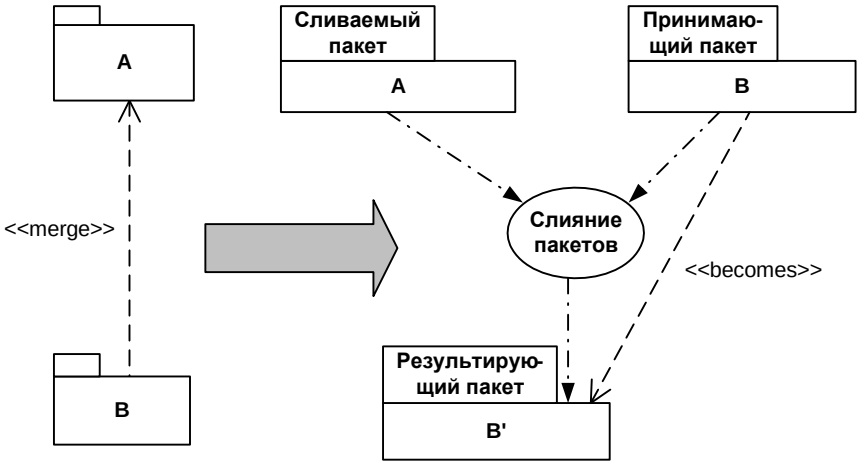


Рис. 6.9. Концептуальное представление семантики слияния пакетов

Общий принцип выполнения слияния состоит в следующем. Любой элемент после слияния не должен обладать меньшими возможностями, чем до выполнения слияния. Это означает, например, что результирующая навигация, кратность или видимость принимающего элемента модели не должны быть уменьшены или исключены в результате слияния пакетов.

Семантика слияния пакетов определяется множеством правил для различных типов элементов в форме ограничений и преобразований. При этом ограничения специфицируют предусловия для допустимого слияния пакетов, в то

время как преобразования описывают допустимые семантические эффекты или постусловия.

Если ограничения правил слияния не выполняются, то слияние пакетов является недопустимым, а содержащая его модель является несостоятельной (ill formed).

Общие правила слияния пакетов

Сливаемый элемент и принимающий элемент соответствуют друг другу, если они удовлетворяют правилам соответствия для их метатипа.

Ограничения:

- на диаграмме пакетов для отношения слияния не должно быть циклов;
- пакет не может сливаться с пакетом, в котором он содержится;
- пакет не может сливаться с пакетом, который он содержит;
- сливаемый элемент, метатип которого не является видом: `Пакет`, `Класс`, `Тип данных`, `Свойство`, `Ассоциация`, `Операция`, `Ограничение`, `Перечисление` или `Перечисление литералов`, не может иметь в качестве принимающего элемента элемент с тем же именем и метатипом, за исключением, когда принимающий элемент является точной копией сливаемого элемента;
- слияние пакетов является допустимым, если и только если все ограничения, требуемые для выполнения слияния, удовлетворены;
- совпадающие типизированные элементы, например, свойства и параметры, должны иметь согласованные типы. Для типов, таких как классы или типы данных, согласованный тип является или тем же типом, или общим супертипом. Во всех других случаях соответствие означает, что типы должны быть одними и теми же;
- принимающий элемент не может иметь явных ссылок ни на один из сливаемых элементов.

Преобразования:

- если для сливаемого или принимающего элемента не существует совпадающего элемента, то они полностью копируются в результирующий пакет. Это правило принимается по умолчанию;
- результатом слияния двух элементов с совпадающими именами и метатипами, которые являются точными копиями друг друга, является принимающий элемент;

- совпадающие элементы объединяются согласно правилам преобразования, специфическим для их метатипа, а результаты включаются в результирующий пакет;
- все ссылки на типизированные элементы, которые оканчиваются в результирующем пакете, преобразовываются в ссылки на соответствующие результирующие элементы, но не на соответствующие им приращения;
- для всех совпадающих элементов: если оба совпадающих элемента имеют закрытую видимость, результирующий элемент должен иметь закрытую видимость, в противном случае результирующий элемент должен иметь общедоступную видимость;
- для всех совпадающих элементов классификаторов: если оба совпадающих элемента являются абстрактными, то результирующий элемент является абстрактным, в противном случае результирующий элемент не является абстрактным;
- для всех совпадающих элементов: если оба совпадающих элемента не являются производными, то результирующий элемент также не является производным, в противном случае результирующий элемент является производным;
- для всех совпадающих элементов кратности: нижняя граница результирующей кратности является меньшей из нижних границ кратности совпадающих элементов;
- для всех совпадающих элементов кратности: верхняя граница результирующей кратности является большей из верхних границ кратности совпадающих элементов;
- любые стереотипы, применяемые к элементу модели или в сливаемом, или в принимающем элементе, являются также применимыми к соответствующему результирующему элементу.

Правила для пакетов

Элементы, которые являются видом `Пакет`, совпадают по имени и метатипу.

Преобразования:

- вложенный пакет из сливаемого пакета преобразовывается во вложенный пакет с тем же именем в результирующем пакете, если только принимающий пакет уже не содержит совпадающий вложенный пакет. В последнем случае сливаемый вложенный пакет рекурсивно сливается с совпадающим принимающим вложенным пакетом;
- импорт элемента, который находится в собственности принимающего пакета, преобразовывается в соответствующий импорт элемента в результи-

рующем пакете. Импортируемые элементы не сливаются, если только это не слияние пакетов в пакет, который владеет импортируемым элементом или его псевдонимом.

Правила для классов и типов данных

Элементы, которые являются видами `Класса` или `Типа данных`, совпадают по имени и метатипу.

Преобразования:

- все свойства сливаемого классификатора сливаются с принимающим классификатором для получения результирующего классификатора согласно правилам преобразования свойств, специфицированных далее;
- вложенные классификаторы сливаются рекурсивно согласно тем же правилам.

Правила для свойств

Элементы, которые являются видами `Свойства`, совпадают по имени и метатипу.

Ограничения:

- как статическая, так и не статическая характеристика совпадающих свойств должна быть одной и той же;
- характеристика уникальности совпадающих свойств должна быть одной и той же;
- любые ограничения, ассоциированные с совпадающими свойствами, не должны быть конфликтующими;
- любые переопределения, ассоциированные с совпадающими свойствами, не должны быть конфликтующими.

Преобразования:

- для сливаемых свойств, которые не имеют совпадающего принимающего свойства, результирующее свойство становится вновь созданным свойством в результирующем классификаторе. Это результирующее свойство является тем же, что и сливаемое свойство;
- для сливаемых свойств, которые имеют совпадающее принимающее свойство, результирующее свойство является свойством с тем же именем и характеристиками, за исключением случая, когда эти характеристики являются различными. Когда эти характеристики являются различными, результирующие характеристики свойства определяются посредством применения соответствующих правил преобразования;

- для совпадающих свойств: если оба свойства являются "только для чтения", результирующее свойство также является "только для чтения". В противном случае результирующее свойство не является "только для чтения";
- для совпадающих свойств: если оба свойства являются неупорядоченными, то результирующее свойство также является неупорядоченным. В противном случае результирующее свойство является упорядоченным;
- для совпадающих свойств: если ни одно из свойств не является подмножеством некоторого производного объединения, то результирующее свойство не назначается как некоторое подмножество. В противном случае результирующее свойство будет назначено как некоторое подмножество этого производного объединения;
- для совпадающих свойств: различные переопределения совпадающих свойств являются объединением конъюнкций;
- для совпадающих свойств: различные ограничения на совпадающие свойства являются объединением конъюнкций;
- для совпадающих свойств: если или сливаемый, и/или принимающий элементы не является уникальными, то результирующий элемент не является уникальным. В противном случае результирующий элемент назначается как уникальный;
- результирующий тип свойства преобразуется, чтобы ссылаться на соответствующий тип в результирующем пакете.

Правила для ассоциаций

Элементы, которые являются видом `Ассоциации`, совпадают по имени и по их концам ассоциаций. Совпадение концов ассоциаций означает совпадение по имени и типу, аналогично правилу для свойств. Совпадение ассоциаций включает случай, когда ассоциации вовсе не имеют имени. Эти правила дополняют описанные ранее правила для свойств.

Ограничения:

- настоящие правила применяются только к бинарным ассоциациям. Для слияния *n*-арных ассоциаций следует использовать общие правила;
- если совпадающий сливаемый конец ассоциации является композитным, то принимающий конец ассоциации должен быть композитным;
- если совпадающий сливаемый конец ассоциации является собственностью этой ассоциации, то принимающий конец ассоциации должен быть в собственности ассоциации.

Преобразования:

- слияние совпадающих ассоциаций выполняется посредством слияния классификаторов ассоциаций с использованием правил слияния для классификаторов, а слияние соответствующих им собственных свойств концов ассоциации выполняется согласно правилам для свойств и концов ассоциаций;
- для совпадающих концов ассоциаций: если ни один из концов ассоциации не является направленным, то результирующий конец ассоциации также не является направленным. Во всех других случаях результирующий конец ассоциации является направленным.

Правила для операций

Элементы, которые являются видом `Операция`, совпадают по имени, порядку параметров и типам параметров, не включая тип возвращаемого значения.

Ограничения:

- параметры операции и типы должны соответствовать таким же правилам для типа и кратности, какие были определены для свойств;
- если совпадающая сливаемая операция является запросом, то принимающая операция должна быть запросом.

Преобразования:

- для сливаемой операции, которая не имеет совпадающей принимающей операции, результирующая операция является операцией с тем же именем и сигнатурой, что и в сливаемом классификаторе;
- для сливаемой операции, которая имеет совпадающую принимающую операцию, результирующая операция является следствием слияния совпадающих сливаемой и принимающей операций. При этом преобразования параметров выполняются согласно правилам для преобразования свойств, которые определены ранее.

Правила для перечислений

Элементы, которые являются видами `Перечисления литералов`, совпадают по собственному перечислению и имени литерала.

Ограничение: совпадающие литералы перечисления должны следовать в одном и том же порядке.

Преобразование: несовпадающие литералы перечисления из сливаемого перечисления конкатенируются в принимающем перечислении.

Правила для ограничений

Ограничение: ограничения должны быть взаимно непротиворечивы.

Преобразования: ограничения на сливаемые элементы модели конъюнктивно добавляются к совпадающим ограничениям в принимающем элементе модели.

Представленные правила слияния могут быть проиллюстрированы следующим фрагментом диаграммы (рис. 6.10), на котором пакеты `P` и `Q` сливаются в пакете `R`, в то время как пакет `Q` сливается только в пакете `S`.

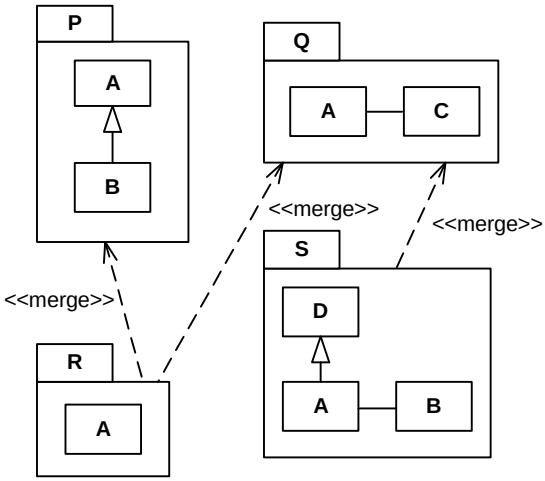


Рис. 6.10. Иллюстрация правил слияния пакетов

Результаты этих слияний в форме преобразованных пакетов `R` и `S` изображены на следующей диаграмме (рис. 6.11). Выражения в квадратных скобках указывают, какие индивидуальные приращения будут слиты для получения конечного результата. При этом символ `@` используется для обозначения оператора слияния, который, вообще говоря, не является частью стандартной нотации языка UML 2.0. Использование этого символа определяется только соображениями удобства.

В спецификации языка UML 2.0 явные преобразования слияния определены только для нескольких общих метатипов. Семантика слияния других метатипов, таких как конечные автоматы и взаимодействия, являются более сложными и могут зависеть от предметной области. Элементы, для которых отсутствуют правила слияния, могут преобразовываться при слиянии согласно правилу по умолчанию: такие элементы полностью копируются в результирующий пакет. Разработчики на практике могут заменить это правило для специальных случаев слияния с помощью профилей или других способов расширения языка UML 2.0.

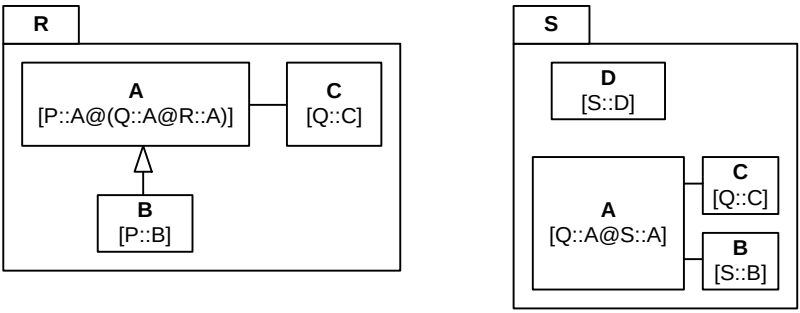


Рис. 6.11. Результаты слияния пакетов

* * *

В заключение описания диаграммы пакетов приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм пакетов (табл. 6.1). Графические узлы и пути, приводимые на диаграммах пакетов, могут изображаться также на других диаграммах структуры.

Таблица 6.1. *Графические узлы и пути на диаграммах пакетов*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Пакет (package)	**Имя пакета**	Пакет *(см. разд. 6.1.1)*
Класс (class)	**Имя класса**	Класс *(см. разд. 4.2)*
Импорт пакета, общедоступный (package import, public)	<<import>>	Импорт пакета *(см. разд. 6.1.3)*
Импорт пакета, закрытый (package import, private)	<<access>>	Импорт пакета *(см. разд. 6.1.3)*
Слияние пакетов (package merge)	<<merge>>	Слияние пакетов *(см. разд. 6.1.4)*

6.2. Диаграмма объектов

Диаграмма объектов (object diagram) предназначена для спецификации объектов и связей между ними для фиксированного момента времени.

Хотя этот тип канонических диаграмм похож на диаграммы коммуникации, однако между ними имеется принципиальное различие — на диаграммах коммуникации присутствуют сообщения, которые отсутствуют на диаграммах объектов. В дополнение к этому понятие объекта в языке UML 2.0 отличается от линии жизни, хотя между ними и есть много общего. Основными элементами, которые изображаются на диаграммах объектов и являются собственно объекты.

6.2.1. Объект

Объект (object) является отдельным экземпляром класса, который создается на этапе реализации модели или выполнения программы.

Каждый объект обладает определенной индивидуальностью, которая предполагает наличие у него уникального имени, отличающегося от имен других возможных объектов этого же класса. Кроме собственного имени объект может иметь и конкретные значения атрибутов. В качестве графического символа объекта используется символ классификатора, внутри которого записываются имя объекта и, возможно, значения его атрибутов.

Имя объекта представляет собой строку текста, записанную в следующем виде (БНФ):

```
<имя-объекта>::= [<собственное-имя-объекта>] | [:<имя-класса>
[','<имя-класса>]*]
```

В записи имени объекта <*собственное-имя-объекта*> и <*имя-класса*> не могут отсутствовать одновременно, а вся запись имени объекта подчеркивается. Если указано <*собственное-имя-объекта*>, то оно должно начинаться со строчной буквы. При наличии <*имени-класса*>, ему всегда должно предшествовать двоеточие.

На диаграммах объектов могут встретиться следующие варианты возможных записей имен объектов:

- ❒ ○: C — для объекта специфицировано собственное имя объекта и имя класса;
- ❒ ○ — для объекта специфицировано только собственное имя объекта;
- ❒ : C — для объекта специфицировано только имя класса.

Рассмотрим примеры нескольких конкретных изображений объектов (рис. 6.12). Полная спецификация имени объекта содержит оба элемента его имени (рис. 6.12, *а*). Если отсутствует имя класса, то такой объект называется *сиротой*. Для него записывается только собственное имя объекта, а двоеточие и имя класса не указываются (рис. 6.12, *б*). Если отсутствует собственное имя объекта, то такой объект принято называть *анонимным*, при этом обязательно записывается двоеточие перед именем соответствующего класса (рис. 6.12, *в*).

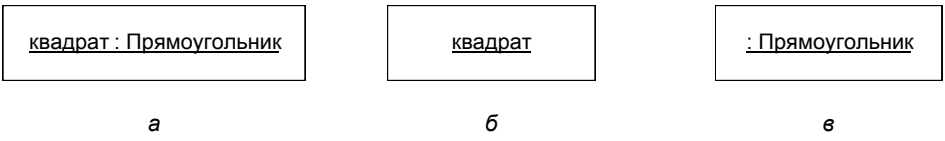


Рис. 6.12. Примеры графических изображений объектов на диаграммах структуры: *а* — полная спецификация имени объекта; *б* — имя класса отсутствует; *в* — отсутствует имя объекта

Промежуточным случаем между классом и объектом является спецификация экземпляра.

6.2.2. Спецификация экземпляра

Спецификация экземпляра (instance specification) является элементом модели, который представляет описание произвольной сущности типа классификатора.

Цель спецификации экземпляра заключается в том, чтобы показать, что некоторая сущность представляет интерес в моделируемой системе. Спецификация экземпляра указывает на существование некоторой сущности в моделируемой системе и полностью или частично описывает эту сущность. Такое описание может включать:

- классификацию этой сущности посредством одного или более классификаторов, сущность которых является экземпляром. Если единственный специфицированный классификатор является абстрактным, то спецификация экземпляра только частично описывает эту сущность;
- вид экземпляра, основанный на его классификаторе или классификаторах. Например, спецификация экземпляра, классификатор которого является классом, описывает некоторый объект этого класса, в то время как спецификация экземпляра, классификатор которого является ассоциацией, описывает связь этой ассоциации;

- спецификацию значений структурных характеристик сущности. Не все структурные характеристики классификаторов спецификации экземпляра могут быть представлены слотами, в некотором случае спецификация экземпляра является частичным описанием;
- необязательную спецификацию того, как вычислять, получать или создавать экземпляр.

В языке UML 2.0 спецификация экземпляра является конкретным классом. Именно поэтому следует различать спецификацию экземпляра и объект, хотя их графические изображения практически идентичны.

Спецификация экземпляра может представлять иллюстрацию или пример потенциально возможной сущности в моделируемой системе. Соответствующие детали спецификации экземпляра могут быть неполными.

Спецификация экземпляра изображается с использованием такого же символа, как его классификатор. Имя спецификации экземпляра представляет собой строку текста, записанную в следующем виде (БНФ):

```
<имя-спецификации-экземпляра>::= [<собственное-имя-спецификации-
экземпляра>] | [:<имя-класса> [','<имя-класса>]*]
```

Вся запись имени спецификации экземпляра подчеркивается. Если указано <*собственное-имя-спецификации-экземпляра*>, то оно должно начинаться со строчной буквы. При указании нескольких имен классификаторов их имена отделяются запятыми. В записи имени спецификации экземпляра <*собственное-имя-спецификации-экземпляра*> и <*имя-класса*> могут отсутствовать одновременно. Отсутствие имени у спецификации экземпляра может отражать его отсутствие в базовой модели. Стандартной нотацией для спецификации анонимного экземпляра безымянного классификатора является подчеркнутое двоеточие (:).

Спецификация экземпляра, классификатором которого является ассоциация, представляется в форме связи и изображается с использованием такой же нотации, как и для ассоциации. В этом случае отрезком непрерывной линии соединяются не классификаторы, а спецификации экземпляров. Подчеркнутое имя связи можно не показывать, если это ясно из самого соединения спецификаций экземпляров. Дополнительно могут быть изображены имена концов для этой связи и стрелки навигации. Но если они показаны, они должны согласовываться с навигацией концов соответствующей ассоциации.

В качестве примера рассмотрим бинарное отношение между спецификациями экземпляров: `Иван` и `Егор` класса `Человек` (рис. 6.13). Они связаны между

собой бинарной связью без имени в форме линии ассоциации. Для концов этой связи указаны имена ролей соответствующих спецификаций экземпляров.

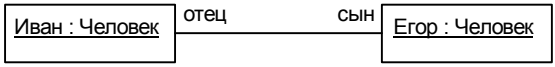


Рис. 6.13. Спецификации экземпляров, которые представляют собой два объекта, соединенные связью

Спецификации экземпляров не описывают особенности времени выполнения объектов соответствующих классификаторов. Они указывают только общую информацию о таких объектах. При этом нельзя сделать никаких выводов относительно деталей реализации классов времени выполнения. При этом спецификации экземпляров могут быть не полными — отдельные характеристики классификаторов могут быть пропущены. Сами классификаторы спецификаций экземпляров могут быть абстрактными, даже если действительные сущности должны иметь конкретных классификаторов.

В отдельных случаях спецификация экземпляра может представлять сущность в некоторой точке времени в форме "моментального снимка". При этом изменение сущности может быть смоделировано с использованием нескольких спецификаций экземпляра, по одному для каждого моментального снимка. Сама сущность должна согласовываться с каждой спецификацией данного экземпляра и иметь характеристики со значениями, указанными в каждом слоте соответствующей спецификации экземпляра.

6.2.3. Слот

Слот (slot) предназначен для представления того, что сущность, моделируемая посредством спецификации экземпляра, имеет конкретное значение или значения для некоторой своей структурной характеристики.

Слот позволяет связать спецификацию экземпляра, структурную характеристику и одно или несколько значений. Слот является собственностью спецификации экземпляра и специфицирует одно или несколько значений для отдельной структурной характеристики спецификации экземпляра, который владеет этим слотом. При этом значения в слоте должны соответствовать характеристике этого слота (по типу, кратности и т. д.).

Спецификация экземпляра может иметь один слот для отдельной структурной характеристики своих классификаторов, включая наследуемые характеристики. Нет необходимости использовать слот для каждой наследуемой

структурной характеристики. Если структурная характеристика описана в классе, то она специфицирует значения, которые могут быть помещены в этот слот.

Слоты изображаются с использованием такой же нотации, что и соответствующие структурные характеристики. Если характеристика изображается в секции атрибутов класса в форме текста, то слот для этой характеристики также может быть изображен текстом в следующем формате (БНФ):

```
<слот>::= <имя-характеристики> [: <тип-характеристики>] '='
<спецификация-значения>
```

Так, например, спецификация экземпляра `мойАдрес` (рис. 6.14, *а*) имеет следующие слоты: `имяУлицы` (тип — строка со значением `"Садовая"`), `номерДома` (тип — целое число со значением `25`) и `номерКвартиры` (тип — целое число со значением `15`). Спецификация анонимного экземпляра класса `Квадрат` (рис. 6.14, *б*) имеет 4 слота без указания их типа. В этом случае тип каждой структурной характеристики может быть описан в соответствующем классе.

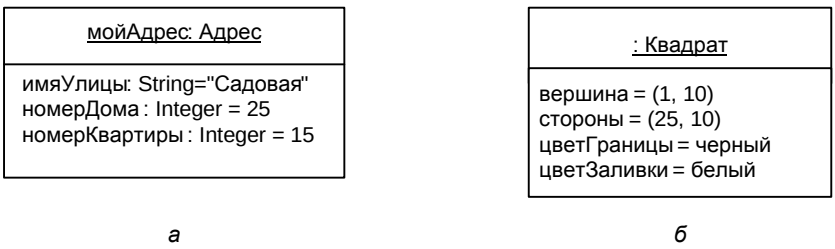


Рис. 6.14. Примеры слотов со значениями: *а* — спецификация экземпляра `мойАдрес`; *б* — спецификация анонимного экземпляра класса `Квадрат`

Характеристика, определяющая каждый слот, может быть непосредственной или наследуемой структурной характеристикой классификатора для рассматриваемой спецификации экземпляра. При этом одна структурная характеристика, включая одну и ту же характеристику, наследуемую от нескольких классификаторов, может являться характеристикой не более одного слота в спецификации экземпляра.

Отсутствие слота для некоторой характеристики в спецификации экземпляра не означает, что представляемая сущность не имеет этой характеристики, но указывает только лишь на то, что эта характеристика не представляет интереса в модели.

Поскольку спецификация экземпляра является классом, она может иметь конкретные значения для этого экземпляра.

6.2.4. Значение экземпляра

Значение экземпляра (instance value) является спецификацией значения, которое идентифицирует некоторый экземпляр.

Спецификация значений представляет собой спецификацию того, как вычислять, получать или создавать экземпляр некоторого классификатора. При этом значение экземпляра специфицирует значение, моделируемое посредством спецификации экземпляра.

Значение экземпляра может представляться с использованием текстовой или графической нотации. Текстовая нотация применяется для указания значения слота атрибута, при этом изображается имя соответствующего экземпляра. Графическая нотация служит для представления значения экземпляра в форме ссылки и связывания ее с соответствующим экземпляром.

Если спецификация экземпляра использует спецификацию значения, то оно может быть изображено либо после имени спецификации экземпляра и знака равенства, либо без знака равенства ниже имени. Если спецификация экземпляра изображается в форме прямоугольника класса, то спецификация значения может быть изображена в этом прямоугольнике в секции имени и ниже его (рис. 6.15).

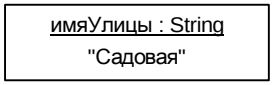
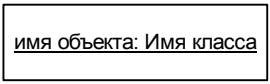
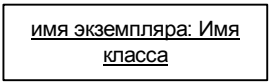


Рис. 6.15. Спецификация экземпляра `имяУлицы`

* * *

В заключение описания диаграммы объектов приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм объектов (табл. 6.2). Графические узлы и пути, изображаемые на диаграммах объектов, могут присутствовать также на других диаграммах структуры.

Таблица 6.2. *Графические узлы и пути на диаграммах объектов*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Объект (object)		Объект *(см. разд. 6.2.1)*
Спецификация экземпляра (instance specification)		Спецификация экземпляра *(см. разд. 6.2.2)*
Связь (link)		Соединитель *(см. разд. 5.2.2)*

Глава 7

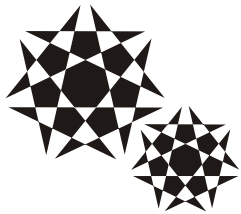


Диаграмма последовательности (sequence diagram)

Рассмотренные ранее диаграммы структуры представляют собой логическую модель статического представления моделируемой системы. На диаграммах классов, композитной структуры, пакетов и объектов могут быть изображены только взаимосвязи структурного характера между элементами модели, которые не зависят от времени или реакции системы на внешние события. Однако различные элементы системы могут оказывать определенное воздействие друг от друга или находиться в определенном взаимодействии.

Для моделирования взаимодействия элементов моделей в языке UML 2.0 используются различные диаграммы *взаимодействия*. Говоря об этих диаграммах, имеют в виду различные способы визуализации взаимодействия. Взаимодействие элементов модели можно рассматривать в контексте статической структуры модели. В этом случае для представления структурных особенностей передачи и приема сообщений между элементами используется диаграмма коммуникации. Взаимодействие элементов можно рассматривать в контексте выполняемых элементами действий. В этом случае для представления структуры выполняемых действий используются диаграммы деятельности и обзора взаимодействия.

Взаимодействие элементов можно рассматривать во времени. Для представления временных особенностей передачи и приема сообщений между элементами могут использоваться диаграммы последовательности и временная диаграмма. Временная диаграмма предназначена для моделирования отдельных синхронных процессов, описывающих взаимодействия элементов с точной привязкой ко времени. Для моделирования синхронных и асинхронных процессов без точной привязки ко времени предназначена диаграмма последовательности, которая является предметом рассмотрения настоящей главы.

7.1. Диаграмма последовательности — основная модель взаимодействия элементов проектируемой системы

Диаграмма последовательности (sequence diagram) — диаграмма, предназначенная для представления взаимодействия между элементами модели программной системы в терминологии линий жизни и сообщений между ними.

Графически диаграмма последовательности имеет два измерения. Одно — слева направо в виде вертикальных линий, каждая из которых соответствует линии жизни отдельного участника взаимодействия. Второе — вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. Реализация взаимодействия моделируется посредством сообщений, которые передаются между различными линиями жизни. Сообщения изображаются в виде стрелок различной формы и образуют некоторый порядок относительно времени своей передачи. При этом сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже.

Масштаб для оси времени на диаграмме последовательности не указывается, поскольку эта диаграмма предназначена для моделирования только лишь временного порядка следования сообщений типа "раньше-позже".

В отличие от других диаграмм взаимодействия, диаграмма последовательности визуализирует порядок следования сообщений по времени, но не отображает структурных отношений между линиями жизни. С помощью диаграммы последовательности можно описать полный контекст взаимодействий как своеобразный временной график "жизни" всей совокупности элементов, взаимодействующих между собой для реализации варианта использования, достижения бизнес-цели или выполнения какой-либо задачи.

Диаграмма последовательности, как правило, используется разработчиками для лучшего понимания особенностей взаимодействия элементов модели в контексте выполнения программного приложения. Разработка диаграммы последовательности осуществляется в ходе фазы детального проектирования, когда должна быть установлена точная коммуникация внутри процесса в соответствии с формальными протоколами. В дальнейшем эта диаграмма может быть использована для тестирования приложения или его отдельных модулей.

Для описания программной системы используется общий механизм моделирования взаимодействий, которые могут быть представлены с различной степенью детализации и ориентированы как на разработчиков программных систем, так и на конечных пользователей и других заинтересованных лиц.

Взаимодействие (interaction) — единица поведения некоторого классификатора, которая концентрирует внимание на наблюдаемом обмене информацией между элементами, являющимися участниками этого взаимодействия.

Графической нотацией для представления взаимодействия на диаграмме последовательности является прямоугольник с непрерывными сторонами, который также называется *фреймом* (frame) диаграммы. В верхнем левом углу прямоугольника фрейма изображается небольшой пятиугольник, в который помещается ключевое слово sd, за которым следует имя взаимодействия и его параметры. Внутри прямоугольника диаграммы изображаются линии жизни и другие фреймы взаимодействия, включенные в данное взаимодействие.

Диаграмма последовательности может включать также определения локальных атрибутов с тем же синтаксисом, что и атрибуты, изображаемые в соответствующей секции символа класса. Эти определения атрибутов могут изображаться рядом с верхней частью фрейма диаграммы или с помощью символов примечания в других частях диаграммы.

Для моделирования деталей взаимодействия и описания последовательности наступления событий в языке UML 2.0 используется специальный термин — *траектория*. Траектория может быть обозначена в форме:

```
<наступление события-1, наступление события-2, ..., наступление события-n>
```

Под *чередующимися* траекториями понимается объединение двух или более траекторий, так что события из различных траекторий могут происходить в любом порядке в результирующей траектории, в то время как события в одной и той же траектории образуют свой собственный порядок. Семантика чередования отличается от ситуации, когда два события могут произойти в точности в одно и то же время.

В общем случае семантика взаимодействия задается как два множества траекторий: допустимых и недопустимых траекторий. Объединение этих двух множеств не обязательно покрывает полный универсум возможных траекторий. Траектории, которые не включены в них, не описываются вовсе этим взаимодействием, и разработчик может не знать, являются ли они допустимыми или нет.

Формально семантика взаимодействия задается парой [P, I], где P есть множество допустимых в действительности траекторий, а I есть множество недопустимых траекторий. Объединение $P \cup I$ не является необходимым для обеспечения полного множества траекторий. Траектория является последовательностью наступления событий, обозначаемых как $<e_1, e_2, \ldots, e_n>$. Наступление событий может также включать информацию о значениях всех существующих объектов в этой точке времени. Каждая конструкция взаимодействия выражается в терминах отношения с этими двумя множествами траекторий. Для простоты обычно ссылаются только на множество допустимых в действительности траекторий, т. к. именно эти траектории на практике моделируются наиболее часто. Два взаимодействия являются *эквивалентными*, если соответствующие им множества траекторий равны.

Предполагается, что моменты наступления событий при моделировании взаимодействия имеют нулевое время. С другой стороны, между наступлениями событий всегда имеется некоторая продолжительность времени. В этом случае любое выполнение определяется в траектории посредством двух наступлений событий, одно в его начале и одно в его конце. В модели выполнения это соответствует наступлению события *начала* и наступлению события *завершения*.

Порядок наступления событий вдоль линий жизни существенен для обозначения последовательности, в которой эти наступления события происходят. Однако абсолютные расстояния между наступлениями событий на линиях жизни не имеют семантики. Другими словами, время на диаграмме последовательности имеет шкалу порядка, а не шкалу отношений, о чем необходимо знать всем разработчикам.

Семантика взаимодействия является композиционной в том смысле, что она строится механически из семантики составляющих его фрагментов взаимодействия. При этом составляющие фрагменты взаимодействия являются упорядоченными и объединяются операцией слабого следования, которая рассматривается далее в этой главе. Множество недопустимых траекторий ассоциируется только с использованием отрицательного комбинированного взаимодействия. Для простоты разработчики часто описывают только допустимые траектории.

Являясь специализацией поведения, взаимодействие обладает возможностью обобщения и переопределения. Специализация взаимодействия может просто добавлять другие траектории к более общему взаимодействию. В этом случае траектории, определенные в специализации, комбинируются с траекториями

исходного взаимодействия в форме объединения. Переопределение взаимодействия означает возможность замены исходного взаимодействия другим взаимодействием.

7.2. Линия жизни

Линия жизни (lifeline) представляет одного индивидуального участника взаимодействия или отдельную взаимодействующую сущность.

В общем случае линия жизни изображается в форме прямоугольника с пунктирной вертикальной линией, идущей от центра прямоугольника вниз (рис. 7.1). Допускается вместо прямоугольника изображать специальный символ классификатора, который представляет эта линия жизни.

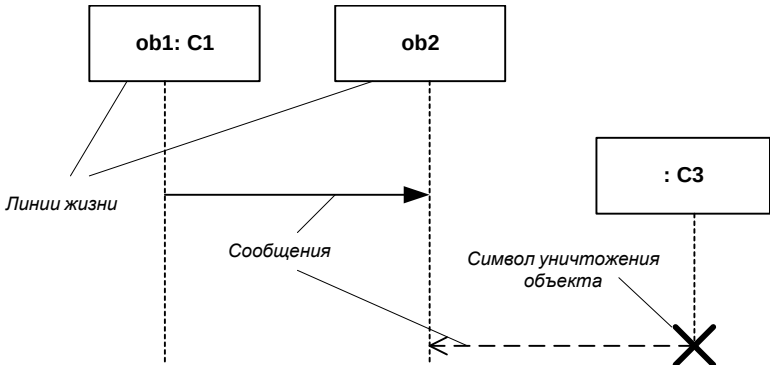


Рис. 7.1. Графическое изображение линий жизни на диаграмме последовательности

Информация, идентифицирующая линию жизни, изображается внутри прямоугольника в следующем формате (БНФ):

```
<идентификатор-линии-жизни>::=
([<имя-связанного-элемента> ['['<селектор>' ] '] ] [: <имя-класса>]
[декомпозиция]) | 'self'
```

где `<селектор>::= <выражение>`

`<декомпозиция>::='ref' <идентификатор-взаимодействия> ['strict']`

Здесь `<имя-класса>` является типом, на который ссылается представленный соединяемый элемент. Если именем является ключевое слово `self`, то такая линия жизни представляет объект классификатора, который владеет данным взаимодействием.

Следует помнить, что, хотя синтаксис позволяет, <*идентификатор-линии-жизни*> не может быть пустым.

Вовсе не обязательно создавать все линии жизни в начальный момент времени. Отдельные линии жизни в системе могут создаваться по мере необходимости, существенно экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник такой линии жизни изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания экземпляра (анонимная линия жизни, созданная от класса C3 на рис. 7.1).

Отдельные линии жизни, выполнив свою роль в системе, могут быть уничтожены или разрушены, чтобы освободить занимаемые ими ресурсы. Для обозначения момента уничтожения линии жизни в языке UML 2.0 используется специальный символ в форме латинской буквы "X" (см. рис. 7.1). Ниже этого символа пунктирная линия не изображается, поскольку соответствующей линии жизни в системе уже нет, и она должна быть исключена из всех последующих взаимодействий. Событие разрушения представляет разрушение экземпляра, описываемого соответствующей линией жизни. Это может привести к последующему разрушению других экземпляров, которыми владеет этот экземпляр посредством композиции.

Если некоторая линия жизни была уничтожена, то вновь возникнуть в системе она уже не может. Вместо него может быть создан другой экземпляр того же класса, который, строго говоря, будет являться другой линией жизни.

В процессе функционирования объектно-ориентированных систем одни линии жизни могут находиться в активном состоянии, непосредственно выполняя определенные действия, или в пассивном состоянии, не выполняя никаких действий и ожидая приема сообщений от других линий жизни. Чтобы явно выделить подобную активность линий жизни, на диаграммах последовательности применяется специальное понятие, получившее название спецификации выполнения.

Спецификация выполнения (execution specification) предназначена для моделирования состояния активности линии жизни в описываемом взаимодействии.

Продолжительность спецификации выполнения представляется двумя спецификациями наступления событий: спецификацией наступления события *начала выполнения* и спецификацией наступления события *окончания выполнения*.

Спецификация выполнения представляется в форме узкого вертикального прямоугольника белого или серого цвета, который перекрывает пунктирную линию соответствующей линии жизни (рис. 7.2). Допускается также представлять спецификацию выполнения более широким горизонтальным прямоугольником с дополнительной меткой, идентифицирующей операцию, которая должна быть выполнена этой линией жизни (`ob3` на рис. 7.2).

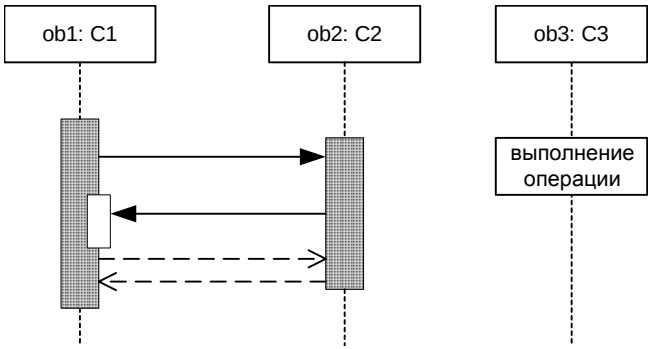


Рис. 7.2. Варианты графического изображения спецификации выполнения

Наступления событий выполнения на одной и той же линии жизни могут перекрываться. В этом случае они могут быть представлены перекрывающимися прямоугольниками (линия жизни `ob1` на рис. 7.2).

7.3. Сообщения и сигналы

В языке UML 2.0 взаимодействие элементов модели рассматривается в информационном аспекте. Информация между взаимодействующими элементами передается в форме сообщений, играющих роль определенных стимулов. В этом случае, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать влияние на поведение своего получателя. Это полностью согласуется с принципами ООАП, когда любые виды информационного взаимодействия между элементами модели должны быть сведены к обмену сообщениями между ними.

Последовательность сообщений рассматривается как важный аспект понимания взаимодействия. Хотя данные, которые передают сообщения и хранят линии жизни, могут играть важную роль в приложении, однако при описании взаимодействий манипуляции с данными не рассматриваются. Сообщения и особенности их спецификации обсуждаются далее в этой главе.

7.3.1. Сообщение

Сообщение (message) — элемент модели, предназначенный для представления отдельной коммуникации между линиями жизни некоторого взаимодействия.

Сообщение специфицирует не только вид коммуникации, но также передатчика и приемника сообщения. Сообщение ассоциируется обычно с двумя спецификациями наступления событий — спецификацией наступления события передачи и спецификацией наступления события приема.

Сообщение моделирует или вызов операции и начало ее выполнения, или передачу и прием некоторого сигнала. Если сообщение представляет вызов операции, то аргументами этого сообщения могут быть как аргументы передающей линии жизни, так и аргументы операции принимающей линии жизни. Если сообщение представляет сигнал, то аргументами этого сообщения могут быть аргументы передающей линии жизни. Если сообщение представляет вызов операции, то обычно изображается некоторое ответное сообщение от вызываемой линии жизни обратно вызывающей линии жизни до того, как эта вызывающая линия жизни продолжит свое дальнейшее поведение.

В общем случае сообщение на диаграмме последовательности изображается в форме линии от передатчика сообщения к приемнику сообщения. Форма этой линии и стрелки зависит от сорта и вида сообщения.

Сорт сообщения (message sort) представляет собой тип перечисления, который идентифицирует характер коммуникации, лежащей в основе генерации данного сообщения.

Сорт сообщения является перечислением следующих значений:

- `'synchCall'` — *синхронное сообщение*, которое соответствует синхронному вызову операции. Синхронные сообщения обычно представляют вызовы методов и изображаются сплошной линией с закрашенной стрелкой (рис. 7.3, *а*);
- `'asynchCall'` — *асинхронное сообщение*, которое соответствует асинхронному вызову операции. Асинхронные сообщения изображаются сплошной линией с открытой стрелкой в форме буквы "V" (рис. 7.3, *б*);
- `'asynchSignal'` — *асинхронный сигнал*, который соответствует некоторому асинхронному действию. Асинхронные сигналы также изображаются сплошной линией с открытой стрелкой в форме буквы "V" (рис. 7.3, *б*).

Сообщение также может быть *ответным* (reply) от вызова метода. Такое сообщение изображается пунктирной линией с открытой стрелкой в форме буквы "V" (рис. 7.3, *в*). Сообщение *создания объекта* (object creation) также изображается пунктирной линией с открытой стрелкой в форме буквы "V" (рис. 7.3, *в*).

Вид сообщения (message kind) представляет собой тип перечисления, который идентифицирует тип сообщения.

Вид сообщения является перечислением следующих значений:

- `'complete'` — *полное сообщение*, для которого существует событие передачи и событие приема. Семантикой полного сообщения является траектория: <Событие передачи, Событие приема>. Сообщения этого вида изображаются рассмотренным ранее образом в зависимости от сорта сообщения;
- `'lost'` — *потерянное сообщение*, для которого существует событие передачи и отсутствует событие приема. Оно интерпретируется как сообщение, которое никогда не достигнет своего места назначения. Семантикой этого сообщения является траектория: <Событие передачи>. Потерянные сообщения изображаются в форме небольшого черного круга на конце стрелки сообщения (рис. 7.3, *г*);
- `'found'` — *найденное сообщение*, для которого существует событие приема и отсутствует событие передачи. Оно интерпретируется как сообщение, инициатор которого находится за пределами области описания. Семантикой найденного сообщения является траектория: <Событие приема>. Это может быть, например, шум или другая деятельность, которую нет необходимости описывать детально. Найденные сообщения изображаются в форме небольшого черного круга на начальном конце сообщения (рис. 7.3, *д*);
- `'unknown'` — *неизвестное* сообщение, для которого отсутствуют событие передачи и событие приема. Эти сообщения не должны представляться на диаграмме последовательности.

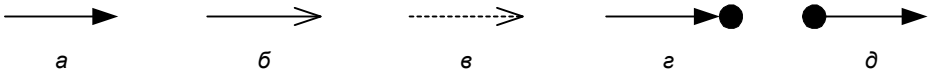


Рис. 7.3. Графическое изображение различных видов сообщений между объектами на диаграмме последовательности

Имя сообщения имеет следующий синтаксис:

```
<идентификатор-сообщения>::= ([<атрибут>'='] <имя-операции-или-сигнала>
['(' [<аргумент> [','<аргумент>]* ')'] [':' <возвращаемое-значение>])|'*'
```

где

```
<аргумент>::= ([<имя-параметра>'='] <значение-аргумента>) |
(<атрибут> '=' <имя-out-параметра> [':' <значение-аргумента>] | '-'
```

Идентификатор сообщения вида `'*'` является сокращением для более сложного альтернативного комбинированного фрагмента с целью представления сообщения любого типа. Возвращаемое значение и назначение атрибута используются только для ответных сообщений.

Назначение атрибута является сокращением для действия, которое назначает некоторое возвращаемое значение данному атрибуту. Оно охватывает как возможное возвращаемое значение сообщения, так и выходные значения параметров **in** и **out**. Если список аргументов содержит только значения аргументов, то все параметры должны соответствовать или этому значению, или тире (-). Если имена параметров используются для идентификации значения аргумента, то аргументы могут быть вообще пропущены. Пропущенным параметрам задают некоторое неизвестное значение аргумента.

Можно привести следующие примеры сообщений:

- `отобразить(10, - , 255, 255, 255, "Привет!")` — сообщение, у которого не определен второй аргумент;
- `длина=возвратить():96` — ответное сообщение, которое передает возвращаемое значение 96, назначаемое атрибуту с именем `длина`;
- `рассчитатьНалог(процент=13)` — сообщение, у которого входному параметру с именем процент задается значение `13`.

Обычно сообщения на диаграмме последовательности изображаются горизонтальными линиями между двумя линии жизни. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий линиями жизни. Если же это предположение не выполняется, то стрелка сообщения изображается под некоторым наклоном, так чтобы конец стрелки располагался ниже ее начала. Заметим также, что сообщения могут иметь шлюзы на одном из своих концов.

Сообщение должно ссылаться или на некоторую операцию (в этом случае сортом сообщения является синхронный или асинхронный вызов операции), или на некоторый сигнал (в этом случае сортом сообщения является асинхронный сигнал). В случае, когда сигнатура сообщения является операцией, аргументы сообщения должны соответствовать параметрам этой операции.

Аргументами сообщения могут быть только следующие элементы:

- атрибуты передающей линии жизни;
- константы;
- символические значения, которые являются произвольными значениями, представляющими любое допустимое значение;
- явные параметры охватываемого взаимодействия;
- атрибуты класса, владеющего данным взаимодействием.

Сообщения не могут пересекать границы комбинированных фрагментов или их операндов.

В дополнение к сорту и виду сообщения могут иметь спецификацию параллельности вызова операции, которая представляется с помощью специального ключевого слова. Вид параллельности вызова операции является перечислением следующих литералов:

- `'sequential'` — вызов *последовательной операции*, для которой не определен механизм управления параллельностью. При параллельных вызовах последовательных операций могут иметь место конфликты. Прием таких сообщений линиями жизни необходимо координировать, так чтобы в каждый момент времени к отдельной операции был направлен только один вызов;
- `'guarded'` — вызов *охраняемой операции*, для которой могут происходить одновременно несколько вызовов операций у одной линии жизни, но допускается к выполнению только один вызов. Другие блокируются до тех пор, пока не закончится выполнение первой операции. В этом случае разработчик системы должен предусмотреть возможность тупиков при одновременных блокировках вызовов;
- `'concurrent'` — вызов *параллельной операции*, для которой могут происходить одновременно несколько вызовов операций у одной линии жизни, и все они могут быть обработаны параллельно.

7.3.2. Сигнал

Сигнал (signal) представляет собой спецификацию асинхронной коммуникации между линиями жизни.

Сигнал может вызвать некоторую реакцию в приемнике при асинхронном способе передачи и без необходимости ответа. Передатчик сигнала не блокируется для ожидания ответа, но продолжает свое выполнение. Классификатор

приемника сигнала должен иметь спецификацию того, как его экземпляры должны принимать этот сигнал и каким поведением реагировать на него.

Поскольку сигнал является классификатором, на диаграмме последовательности изображаются экземпляры сигналов.

Принимающая линия жизни обрабатывает экземпляр сигнала согласно спецификации своего классификатора. Данные, переносимые сигналом, представляются в форме атрибутов экземпляра этого сигнала. При этом сигнал может быть определен независимо от классификатора, специфицирующего обработку этого сигнала. Детали обработки сигнала специфицируются поведением, ассоциированным с этим приемом или самим классификатором. Например, прием сигнала может служить причиной срабатывания триггерных переходов конечных автоматов, которые рассматриваются далее в *главе 10*.

Прием экземпляра асинхронного сигнала представляет собой специальное понятие языка UML 2.0 — событие сигнала.

Событие сигнала (signal event) представляет собой прием линией жизни некоторого асинхронного сигнала.

Событие сигнала происходит, когда сообщение в форме сигнала, первоначально возникающее в результате действия передачи, выполняемого некоторым объектом, принимается другим (возможно тем же) объектом. Событие сигнала может привести к выполнению поведения, которое реализует прием, совпадающий с принимаемым сигналом. Спецификация события сигнала обозначается с использованием следующего формата (БНФ) :

<*событие-сигнала*>::= <*имя-сигнала*> ['(' [<*спецификация-назначения*>] ')']

где

<*спецификация-назначения*>::= <*имя-атрибута*> [',' <*имя-атрибута*>]*

Здесь <*имя-атрибута*> является неявным назначением соответствующего параметра сигнала некоторому атрибуту с этим же именем из контекста классификатора, владеющего соответствующим поведением. Элемент <*спецификация-назначения*> является необязательным и может быть пропущен, даже если сигнал имеет параметры.

Если сигнатура сообщения является сигналом, аргументы сообщения должны соответствовать атрибутам этого сигнала. Аргумент сообщения соответствует атрибуту сигнала, если этот аргумент относится к одному и тому же классу или является специализацией этого атрибута.

Можно привести следующие примеры сигналов:

- `mySignal(s: String)` — сигнал со строковым параметром `s`;
- `оповеститьКлиента` — сигнал без параметров.

Событие сигнала может служить причиной некоторой реакции, как это специфицировано в поведении классификатора, от которого инстанцирован принимающий объект или линия жизни. Другими словами, любое событие сигнала должно быть специфицировано в форме сигнала, который находится в собственности или наследуется от классификатора, который указан в качестве принимающего объекта.

7.4. Комбинированный фрагмент

Комбинированные фрагменты в языке UML 2.0 являются мощным механизмом моделирования вложенного поведения, с помощью которого разработчик может представить несколько траекторий взаимодействия в компактной для восприятия форме. В общем случае для представления отдельной части взаимодействия используются фрагменты взаимодействия.

Фрагмент взаимодействия (interaction fragment) является абстрактным понятием для представления общей единицы или части взаимодействия.

Каждый фрагмент взаимодействия концептуально похож на само взаимодействие. В метамодели языка UML 2.0 фрагмент взаимодействия является абстрактным классом, семантика которого определяется парой множеств траекторий, аналогично взаимодействию. На диаграммах последовательности фрагменты взаимодействия являются упорядоченными согласно их геометрической позиции по вертикали. Геометрическая позиция фрагмента взаимодействия задается самой верхней вертикальной координатой входящих в него сообщений. Частным случаем фрагмента взаимодействия является комбинированный фрагмент, для которого может быть дополнительно специфицирована семантика выполнения его поведения.

Комбинированный фрагмент (combined fragment) — элемент модели, предназначенный для представления внутренней логической структуры фрагментов взаимодействия.

Комбинированный фрагмент определяется посредством оператора взаимодействия и соответствующих ему операндов взаимодействия. Семантика

комбинированного фрагмента зависит от оператора взаимодействия, возможные значения которого рассматриваются далее в этой главе.

Операнд взаимодействия (interaction operand) — отдельный фрагмент взаимодействия, предназначенный для использования в качестве внутренней части комбинированного фрагмента.

Операнд взаимодействия содержится в комбинированном фрагменте и может иметь необязательное сторожевое условие. В этом случае для получения траекторий соответствующего комбинированного фрагмента могут рассматриваться только такие операнды взаимодействия, сторожевые условия которых оцениваются как "истина" в заданной точке взаимодействия. Операнд взаимодействия, в свою очередь, может содержать любое упорядоченное множество произвольных фрагментов взаимодействия.

Если операнд взаимодействия не содержит сторожевого условия, то по умолчанию это означает, что сторожевое условие всегда принимает значение "истина". Семантика операнда взаимодействия задается составляющими его фрагментами взаимодействия, объединенными неявно оператором **seq**, который описывается далее в этом разделе.

Операнды взаимодействия в комбинированном фрагменте отделяются друг от друга пунктирными горизонтальными линиями. Операнды взаимодействия совместно образуют *фрейм* (frame) комбинированного фрагмента. В каждом операнде взаимодействия на диаграмме последовательности порядок фрагментов взаимодействия задается сверху вниз. Если операнд взаимодействия имеет сторожевое условие, то оно должно быть размещено выше любого сообщения, которое является первым сообщением или событием в данном операнде взаимодействия. При этом само сторожевое условие должно содержать только ссылки на значения, локальные к линии жизни, на которой оно находится, или значения, глобальные к моделируемому взаимодействию в целом.

Если оператором взаимодействия является **opt**, **loop**, **break** или **neg**, то комбинированный фрагмент должен содержать в точности один операнд.

Графическое обозначение комбинированного фрагмента на диаграмме последовательности представляет собой прямоугольник со сплошной линией сторон. Оператор взаимодействия изображается в пятиугольнике, располагаемом в верхнем левом углу прямоугольника. Операнды комбинированного фрагмента изображаются вложенными в прямоугольник соответствующего комбинированного фрагмента с использованием пунктирных горизонтальных

линий для разделения на регионы, соответствующие этим отдельным операндам (рис. 7.4).

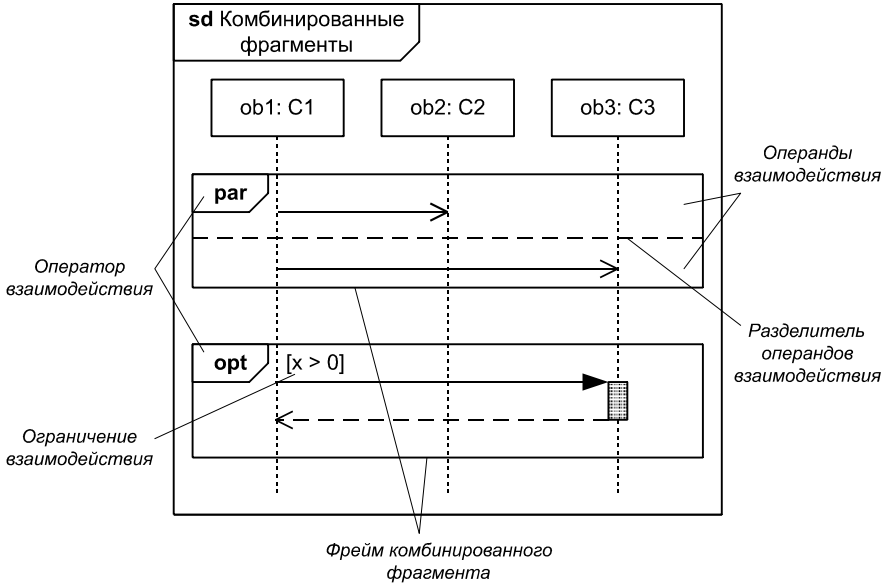


Рис. 7.4. Графическое изображение комбинированного фрагмента

В пятиугольнике фрейма можно указывать больше одного оператора, что соответствует сокращенному обозначению вложенных комбинированных фрагментов. Так, например, **`sd strict`** в имени фрейма означает то же самое, что и два комбинированных вложенных фрагмента, внешний из которых **`sd`**, а внутренний — **`strict`**.

Ограничение взаимодействия (interaction constraint) представляет собой логическое выражение, которое выступает в роли сторожевого условия некоторого операнда в комбинированном фрагменте.

Ограничение взаимодействия является специализацией общего ограничения. Кроме этого, ограничение взаимодействия может содержать два выражения, обозначающие минимальное и максимальное количество раз, которые должен исполняться комбинированный фрагмент **`loop`** (цикл).

Ограничение взаимодействия изображается в квадратных скобках на линии жизни выше первого сообщения или события в соответствующем фрагменте

или операнде взаимодействия. Текст ограничения взаимодействия имеет следующий синтаксис:

```
<ограничение взаимодействия>::= ['['(<логическое-выражение> | 'else') ']']
```

Если ограничение взаимодействия пропущено, то по умолчанию предполагается значение "истина".

Оператор взаимодействия (interaction operator) определяет тип комбинированного фрагмента и является перечислением следующих 12 литералов: **alt**, **assert**, **break**, **critical**, **ignore**, **consider**, **loop**, **neg**, **opt**, **par**, **seq**, **strict**. Семантика этих операторов рассматривается далее в отдельных разделах этой главы.

7.4.1. Альтернативы (*alt*)

Оператор взаимодействия **alt** специфицирует комбинированный фрагмент *Альтернативы* (alternatives), который представляет некоторый выбор поведения. При этом выбор может быть сделан не более одного из операндов. Выбранный операнд должен иметь явное или неявное выражение сторожевого условия, которое в этой точке взаимодействия должно принимать значение "истина". Если операнд не имеет никакого сторожевого условия, то неявно предполагается, что сторожевое условие имеет значение "истина".

Множество траекторий, которые являются возможными при выборе, является объединением траекторий операндов и соответствующих сторожевых условий. Операнд, помеченный сторожевым условием `[else]`, обозначает отрицание дизъюнкции всех других сторожевых условий этого комбинированного фрагмента. Другими словами, оно является истинным, если ни одно другое сторожевое условие не является истинным.

Если ни один из операндов не имеет сторожевого условия, которое оценивается как "истина", то не выполняется ни один из операндов этого комбинированного фрагмента. В этом случае весь комбинированный фрагмент пропускается и выполняется следующий за ним фрагмент или оставшаяся часть описываемого взаимодействия.

В качестве примера рассмотрим диаграмму последовательности для взаимодействия `Покупка товара`, которая содержит комбинированный фрагмент *Альтернативы* с двумя операндами (рис. 7.5). Если выполняется сторожевое условие `[кредит не превышен]`, то будет выполнен первый операнд, в результате чего будет создана линия жизни от класса `Транзакция` с последующим вызовом операции с именем `оплатитьПоКарте()`. При этом второй операнд будет пропущен, и следующим выполнением будет передача сообщения с именем `оформитьДоставку()`.

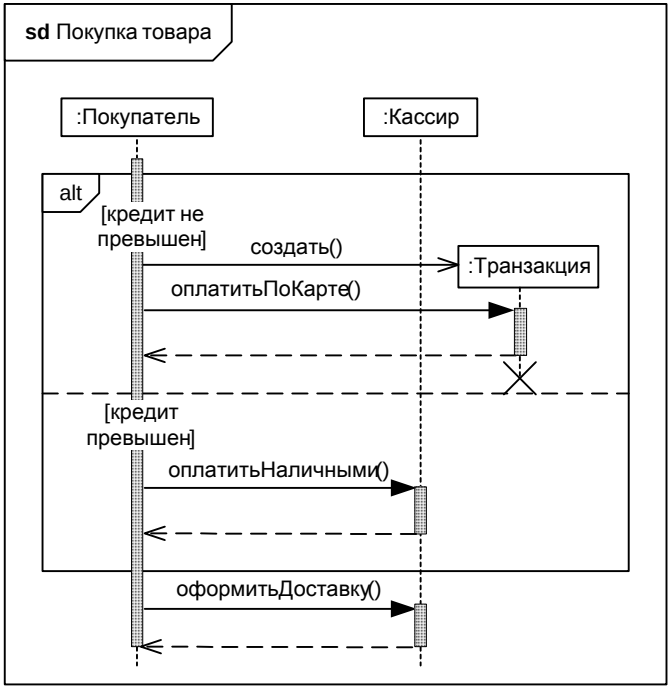


Рис. 7.5. Пример комбинированного фрагмента *Альтернативы*

Если же сторожевое условие [кредит не превышен] не выполняется, то первый операнд будет пропущен, после чего должно быть проверено следующее сторожевое условие — кредит превышен. Если оно выполняется, то будет выполнен второй операнд, в результате чего будет вызвана операция с именем оплатитьНаличными(). После этого будет выполнена передача сообщения с именем оформитьДоставку().

Следует заметить, что сообщение с именем создать является событием сигнала. В этом случае сигнал с тем же именем должен быть описан в классе Транзакция, который служит классификатором для соответствующей линии жизни.

7.4.2. Утверждение (*assert*)

Оператор взаимодействия **assert** специфицирует комбинированный фрагмент *Утверждение* (assertion), который представляет некоторое утверждение. В этом случае единственными следствиями, которые имеют возможность продолжения, являются сообщения или вложенные фреймы данного

операнда. Все другие продолжения приводят в результате к недействительным траекториям. Комбинированный фрагмент `Утверждение` часто объединяется с операторами **ignore** или **consider**, которые рассматриваются далее.

7.4.3. Завершение (*break*)

Оператор взаимодействия **break** специфицирует комбинированный фрагмент *Завершение* (break), который представляет некоторый сценарий завершения. Этот сценарий выполняется вместо оставшейся части фрагмента взаимодействия, который содержит этот соответствующий операнд.

Обычно оператор `Завершение` содержит некоторое сторожевое условие. Если это сторожевое условие принимает значение "истина", то выполняется комбинированный фрагмент *Завершение*, а оставшаяся часть фрагмента взаимодействия, содержащего этот операнд, игнорируется. Если сторожевое условие операнда `Завершение` принимает значение "ложь", то операнд `Завершение` игнорируется, и выполняется оставшаяся часть фрагмента взаимодействия, содержащего этот операнд.

В случае отсутствия сторожевого условия у операнда `Завершение` выбор между выполнением этого операнда и оставшейся частью охватывающего фрагмента взаимодействия выполняется недетерминированно. Операнд `Завершение` должен охватывать все линии жизни фрагмента взаимодействия, который содержит этот оператор.

В качестве примера рассмотрим фрагмент диаграммы последовательности для взаимодействия `Снятие наличных`, которая содержит комбинированный фрагмент *Завершение* (рис. 7.6). Если выполняется сторожевое условие `[ПИН-код неверный]`, то будет выполнен операнд этого фрагмента, в результате чего будет вызвана операции с именем `завершитьТранзакцию()`. При этом оставшаяся часть диаграммы будет пропущена. Если сторожевое условие не выполняется, то операнд `Завершения` будет пропущен, и следующим будет сообщение с именем `предоставитьМенюОпций()`.

Оператор `Завершения` можно считать сокращением оператора `Альтернативы`, когда один операнд задан, а другой полагают равным остальной части рассматриваемого фрагмента взаимодействия. При этом комбинированные фрагменты `Завершения` должны быть глобальными относительно рассматриваемого фрагмента взаимодействия.

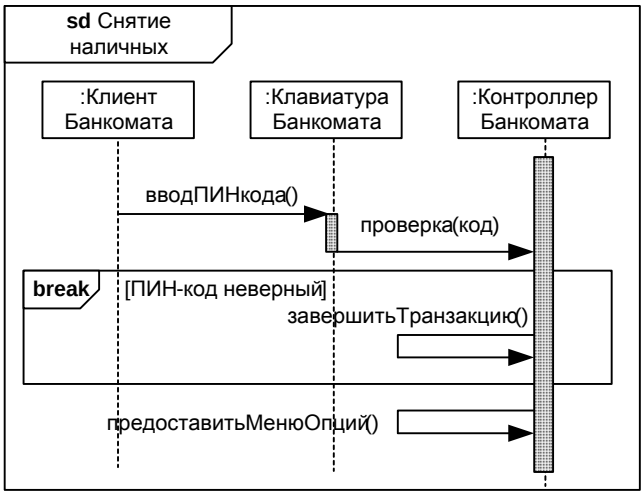


Рис. 7.6. Пример комбинированного фрагмента *Завершение*

7.4.4. Критический регион (*critical*)

Оператор взаимодействия **`critical`** специфицирует комбинированный фрагмент *Критический регион* (critical region), траектории которого не могут чередоваться с другими спецификациями наступления событий на тех линиях жизни, которые этот регион покрывает.

Критический регион рассматривается как неделимый при определении множества возможных траекторий диаграммы или региона, который его содержит. Это означает, что наступления событий или передача сообщений в этом регионе не может чередоваться с наступлениями событий или передачами сообщений в других регионах, даже если другие операторы взаимодействия могут допускать такое чередование. Таким образом, множество траекторий критического региона не может прерываться другими событиями, происходящими вне этого региона.

На практике *Критический регион* используется, как правило, совместно с оператором параллельности, который рассматривается далее. В качестве примера *Критического региона* можно рассмотреть фрагмент диаграммы последовательности, которая моделирует обработку запросов некоторой СУБД (рис. 7.7). Здесь все запросы могут обрабатываться параллельно, причем запросы от разных клиентов и их обработка могут чередоваться между собой. Если же поступает запрос на обновление, который изображен в критическом регионе, то он должен быть сразу обработан. При этом между поступлением запроса на обновление и его обработкой СУБД не могут быть выполнены никакие другие запросы от клиентов.

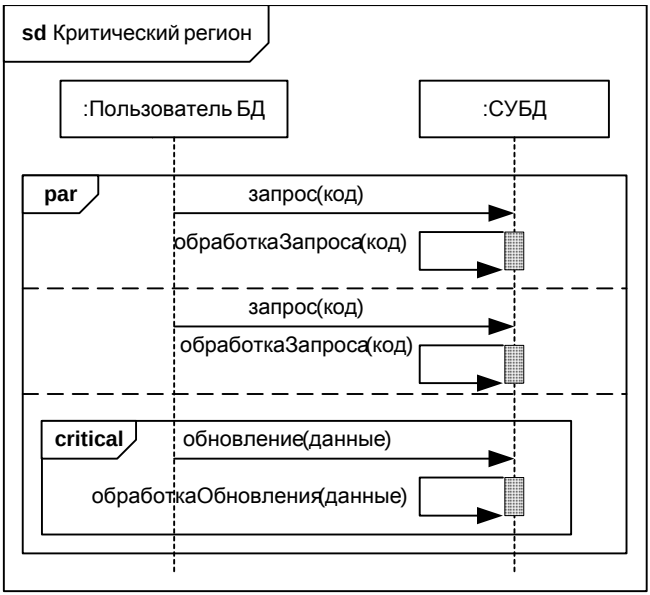


Рис. 7.7. Пример комбинированного фрагмента *Критический регион*

7.4.5. Рассмотрение (*consider*)

Оператор взаимодействия **consider** специфицирует комбинированный фрагмент *Рассмотрение* (consider), в котором изображены только те типы сообщений, какие должны рассматриваться в этом фрагменте. Это эквивалентно определению того, что при рассмотрении данного фрагмента игнорируются любые другие сообщения, которые не изображены в этом фрагменте.

Для фрагмента *Рассмотрение* используется нотация фрейма с оператором, в качестве которого применяется ключевое слово **consider**. Список сообщений следует за операндом и заключается в фигурные скобки согласно следующему формату:

```
<оператор-рассмотрение>::= 'consider''{'<имя-сообщения>[',
'<имя-сообщения>]*'}'
```

Например, выражение **consider** {m, s} указывает, что в соответствующем фрагменте только сообщения m и s рассматриваются как существенные, а все остальные могут быть проигнорированы.

7.4.6. Игнорирование (*ignore*)

Оператор взаимодействия **ignore** специфицирует комбинированный фрагмент *Игнорирование* (ignore), в котором имеются некоторые типы сообщений,

не изображенные на данной диаграмме. Эти типы сообщений рассматриваются как несущественные и могут появляться в траекториях при выполнении соответствующего фрагмента.

Для фрагмента *Игнорирование* используется нотация фрейма с оператором, в качестве которого выступает ключевое слово **ignore**. Список сообщений следует за операндом и заключается в фигурные скобки согласно следующему формату:

```
<оператор-игнорирование>::= 'ignore''{'<имя-сообщения>[',' 
<имя-сообщения>]*'}'
```

Например, выражение **ignore** {q, r} указывает, что в соответствующем фрагменте только сообщения q и r рассматриваются как несущественные.

В качестве примера рассмотрим фрагмент диаграммы последовательности для взаимодействия Снятие наличных, которая содержит комбинированный фрагмент *Игнорирование* (рис. 7.8). В этом фрагменте не рассматриваются сообщения с именами create и destroy вне зависимости от того, к каким линиям жизни они могут быть направлены.

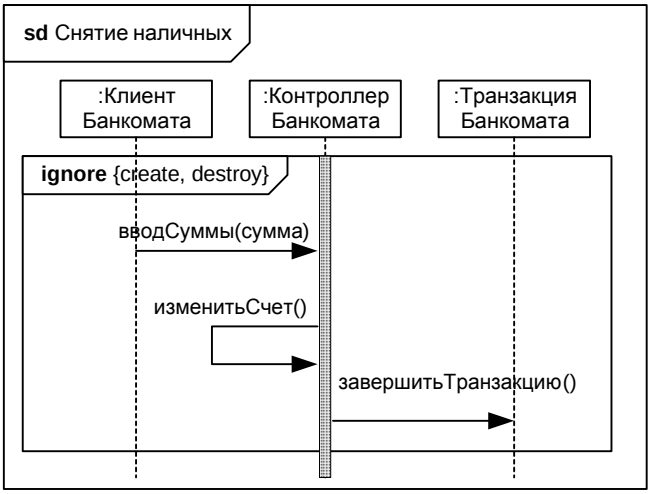


Рис. 7.8. Пример комбинированного фрагмента *Игнорирование*

Следует заметить, что операнды **ignore** и **consider** могут быть объединены с другими операндами в одном прямоугольнике в качестве сокращения для вложенных фреймов. Например, **assert consider** {m, s}, **neg ignore** {q, r}.

7.4.7. Цикл (*loop*)

Оператор взаимодействия **loop** специфицирует комбинированный фрагмент *Цикл* (loop), который представляет собой циклическое повторение некоторой последовательности сообщений. Другими словами, операнд цикла повторяется несколько раз. Дополнительное сторожевое условие может включать нижний и верхний пределы числа повторений цикла, а также некоторое логическое выражение.

Оператор цикла имеет следующий синтаксис (БНФ):

```
<цикл>::='loop'['(' <minint> [',' <maxint> ] ')']
```

где `<minint>::=` неотрицательное натуральное число, которое обозначает минимальное количество итераций цикла; `<maxint>::=` натуральное число, которое обозначает максимальное количество итераций цикла. Значение `<maxint>` должно быть больше или равно `<minint> | '*'`. Здесь символ `*` означает бесконечность.

В общем случае семантика цикла следующая. Операнд цикла всегда повторяется минимальное число раз, которое равно значению `<minint>`. После того как минимальное число повторений будет выполнено, проверяется логическое выражение сторожевого условия. Если это логическое выражение принимает значение "ложь", то выполнение цикла на этом заканчивается. Если же логическое выражение принимает значение "истина", а количество выполненных итераций не превышает значения `<maxint>`, то происходит еще одно выполнение цикла. После этого снова проверяется логическое выражение сторожевого условия, аналогично процедуре выполнения минимального числа повторений.

Другими словами, выполнение цикла будет продолжаться до тех пор, пока логическое выражение будет истинным, но в любом случае не менее минимального и не более максимального количества раз. Если в операторе взаимодействия **loop** специфицировано только значение `<minint>`, то это означает, что `<minint>=<maxint>`. Если специфицируется `<maxint>`, то должно быть специфицировано и значение `<minint>`, при этом значение `<maxint>` должно быть больше или равно значению `<minint>`. Если указан только оператор **loop**, то это означает цикл с бесконечной верхней границей и с 0 в качестве нижней границы.

В качестве примера рассмотрим фрагмент диаграммы последовательности для взаимодействия `Снятие наличных`, которая содержит комбинированный фрагмент *Цикл* (рис. 7.9). Этот цикл будет выполнен по крайней мере один раз, поэтому при первом проходе сторожевое условие не проверяется. Цикл завершается после первого успешного ввода ПИН-кода или после

трех неудачных попыток. При выполнении проверки в качестве результата возвращается булево значение аргумента `result`, которое используется в качестве левого операнда логического выражения сторожевого условия.

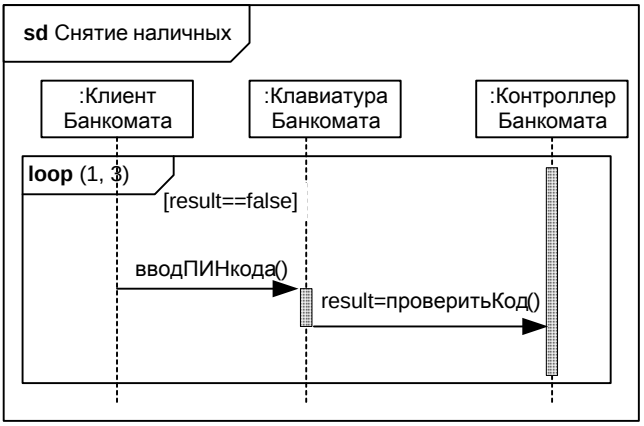


Рис. 7.9. Пример комбинированного фрагмента *Цикл*

Следует заметить, что конструкция цикла представляет рекурсивное применение оператора **seq**, где операнд цикла является следствием результата предыдущих итераций.

7.4.8. Отрицание (*neg*)

Оператор взаимодействия **neg** специфицирует комбинированный фрагмент *Отрицание* (negative), представляющий траектории, которые определяются как недействительные или недопустимые. Множество траекторий, которые определяют комбинированный фрагмент с оператором взаимодействия **neg**, равно множеству траекторий, заданных его единственным операндом. При этом в это множество входят только недействительные или запрещенные траектории. Все фрагменты взаимодействия, кроме *Отрицания*, рассматриваются в положительном смысле, т. е. они описывают траектории, которые являются допустимыми и возможными.

В качестве оператора комбинированного фрагмента *Отрицание* используется ключевое слово **neg**. Фрагмент *Отрицание* может быть полезен в том случае, когда общая модель взаимодействия является неполной, однако некоторые последовательности должны быть запрещены явно.

В качестве примера рассмотрим фрагмент диаграммы последовательности для взаимодействия `Снятие наличных`, которая содержит комбинированный фрагмент *Отрицание*, вложенный в рассматриваемый далее комбинированный фрагмент *Необязательный* (рис. 7.10). Последний будет выполнен только в том случае, когда будет истинно сторожевое условие, т. е. введенный клиентом ПИН-код неверный. В этом случае сообщение с именем `предоставитьМенюОпций()` является недопустимым или запрещенным.

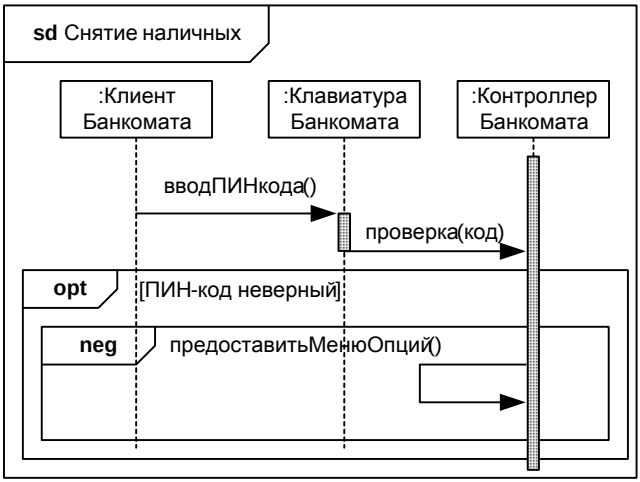


Рис. 7.10. Пример комбинированного фрагмента *Отрицание*

7.4.9. Необязательный (*opt*)

Оператор взаимодействия **`opt`** специфицирует комбинированный фрагмент *Необязательный* (option), который представляет выбор поведения, когда либо выполняется единственный операнд, либо вовсе ничего не выполняется. Оператор выбора семантически эквивалентен альтернативному комбинированному фрагменту, в котором имеется один операнд с непустым содержанием, а второй операнд отсутствует.

Необязательный комбинированный фрагмент состоит из одного операнда со сторожевым условием. Операнд выполняется, если выполнено сторожевое условие. В противном случае операнд не выполняется. Пример комбинированного фрагмента *Необязательный* был рассмотрен ранее (см. рис. 7.10).

7.4.10. Параллельный (*par*)

Оператор взаимодействия **par** специфицирует комбинированный фрагмент *Параллельный* (parallel), который представляет некоторое параллельное выполнение взаимодействий своих операндов. Наступление событий у различных операндов могут чередоваться во времени произвольным образом. Однако внутри каждого операнда соблюдается порядок следования сообщений. Каждый операнд изображается в отдельном регионе, который отделяется от других регионов пунктирной линией.

Хотя на диаграммах последовательности неявно присутствует вертикальная ось времени, ее масштаб у разных линий жизни может быть различным. Потому из относительного положения сообщений по вертикали в различных регионах операндов нельзя сделать никакого заключения об их порядке во времени. В то же время сообщения внутри каждого операнда упорядочены сверху вниз. При этом события, предшествующие отправке сообщения на одной линии жизни, предшествуют событиям, следующим за получением этого сообщения на другой линии жизни.

Пример комбинированного фрагмента *Параллельный* был рассмотрен ранее (см. рис. 7.7). В этом примере может быть передан запрос в первом операнде, затем передан запрос во втором операнде. После чего обработан запрос во втором операнде и, наконец, обработан запрос в первом операнде.

7.4.11. Слабое следование (*seq*)

Оператор взаимодействия **seq** специфицирует комбинированный фрагмент *Слабое следование* (weak sequencing), который состоит из нескольких операндов и представляет слабое следование поведений отдельных операндов. Слабое следование определяется как множество траекторий со следующими свойствами:

- порядок наступления событий в пределах каждого из операндов определяется порядком передачи сообщений во времени (сверху вниз);
- наступление событий на различных линиях жизни у разных операндов могут происходить в произвольном порядке;
- наступление событий на одной линии жизни у различных операндов упорядочиваются сверху вниз, т. е. наступление событий у первого операнда происходит до наступления событий у второго операнда и т. д.

Таким образом, слабое следование сводится к параллельному выполнению, когда операнды находятся на непересекающихся подмножествах линий жизни. Слабое следование сводится к строгому следованию, когда операнды относятся только к одной линии жизни.

7.4.12. Строгое следование (*strict*)

Оператор взаимодействия **`strict`** специфицирует комбинированный фрагмент *Строгое следование* (strict sequencing), который состоит из нескольких операндов и представляет строгий порядок следования поведений отдельных операндов. Данный оператор указывает, что операнды верхнего уровня комбинированного фрагмента выполняются в строго определенном порядке (сверху вниз) и не перекрываются.

Строгий порядок следования означает, что вертикальная координата вложенных фрагментов имеет значение на всем протяжении границ комбинированного фрагмента, а не только для одной линия жизни. Вертикальная позиция спецификации наступления события задается вертикальной позицией соответствующей точки. Вертикальная позиция других фрагментов взаимодействия задается самой верхней вертикальной позицией соответствующих фреймов.

Следует заметить, что операторы слабого и строгого следования на практике используются в контексте вложенных фреймов. Эти конструкции, а также специальные фрагменты и элементы взаимодействия рассматриваются далее в этой главе.

7.5. Специальные фрагменты и элементы взаимодействия

7.5.1. Использование взаимодействия

Использование взаимодействия (interaction use) — элемент модели, представляющий параметризованную ссылку на некоторое взаимодействие в контексте другого взаимодействия.

Использование взаимодействия дает возможность вызывать некоторое поведение в различных взаимодействиях. Выполнение использования взаимодействия эквивалентно выполнению взаимодействия, на которое делается ссылка, с подстановкой действительных значений аргументов, указанных в спецификации использования взаимодействия. В этом случае неявно предполагается, что поведение использования взаимодействия должно быть представлено на отдельной диаграмме последовательности.

Множество траекторий использования взаимодействия является подмножеством траекторий содержащего его взаимодействия. Использование взаимодействия изображается в форме фрейма комбинированного фрагмента с опе-

ратором **ref**, за которым следует полное имя использования взаимодействия. Синтаксис полного имени использования взаимодействия следующий (БНФ):

```
<имя>::= [<имя-атрибута>'='] [<использование-кооперации>'.']
<имя-взаимодействия> ['(' <io-аргумент> [',' <io-аргумент>]* ')'] [':'
<возвращаемое-значение>
```

где `<io-аргумент>::= <in-аргумент> |'out' <out-аргумент>`; `<имя-атрибута>` — атрибут некоторой линии жизни взаимодействия; `<использование-кооперации>` является спецификацией использования некоторой кооперации с линиями жизни данного взаимодействия. Имя взаимодействия в этом случае находится внутри этой кооперации.

В качестве примера рассмотрим фрагмент диаграммы последовательности для взаимодействия Снятие наличных, которая содержит два использования взаимодействия, вложенные в комбинированный фрагмент *Альтернативы* (рис. 7.11). Первое из них ссылается на взаимодействие Выполнение транзакции, которое будет вызвано только в том случае, когда будет истинно первое сторожевое условие, т. е. введенный клиентом ПИН-код является верным. Второе из них ссылается на взаимодействие `Завершение транзакции`, которое будет вызвано только в том случае, когда будет истинно второе сторожевое условие, т. е. введенный клиентом ПИН-код будет неверным.

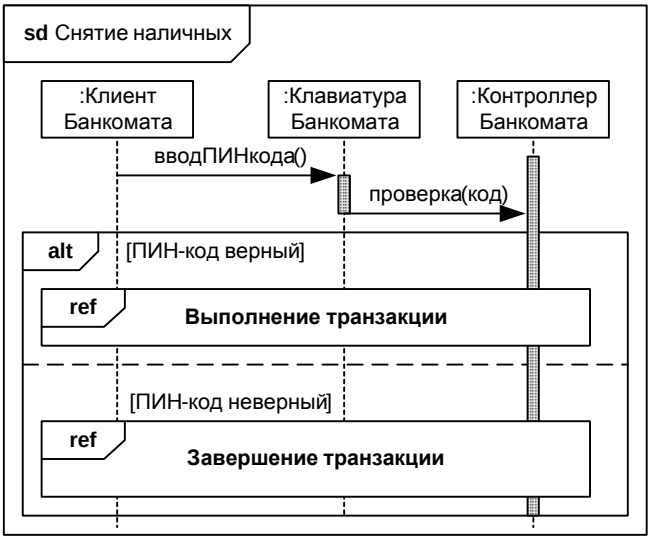


Рис. 7.11. Пример использования взаимодействия

При использовании взаимодействий необходимо соблюдать следующее ограничение: использование взаимодействия должно покрывать все линии

жизни содержащего его взаимодействия. Эти линии жизни должны также присутствовать на диаграмме этого взаимодействия.

7.5.2. Декомпозиция части

Декомпозиция части (part decomposition) является элементом модели, предназначенным для представления внутренних взаимодействий одной из линий жизни, класс которой имеет собственную композитную структуру.

Линия жизни соответствует некоторому классу, который, в свою очередь, может иметь собственную внутреннюю структуру. В этом случае декомпозиция части описывает поведение этой внутренней структуры относительно взаимодействия, которое ссылается на эту декомпозицию. Декомпозиция части является специализацией использования взаимодействия и ассоциируется с линией жизни, класс которой она декомпозирует.

Декомпозиция части обозначается посредством ссылки в заголовке линии жизни на некоторое использование взаимодействия с помощью оператора **`ref`**. Эта конструкция уже встречалась ранее при рассмотрении синтаксиса записи имени линии жизни *(см. разд. 7.2)*. Границы символов фреймов глобальных комбинированных фрагментов изображаются выходящими за пределы границ декомпозиции части. Сообщения, которые входят или выходят из декомпозированной линии жизни, интерпретируются как действительные шлюзы, которые должны соответствовать формальным шлюзам этой декомпозиции.

В качестве примера декомпозиции части рассмотрим фрагмент диаграммы последовательности `Снятие наличных` (рис. 7.12). Здесь линия жизни класса `Банкомат` имеет декомпозицию части, на которую она ссылается в форме `ATM_Снятие`. На этом уровне представления внутренняя структура данного взаимодействия может быть несущественна или неизвестна данному разработчику.

В последующем декомпозиция части может быть представлена в форме отдельной диаграммы последовательности (рис. 7.13), на которой представлены все линии жизни, образующие внутреннюю структуру банкомата. Поскольку комбинированный фрагмент *Альтернативы* является глобальным, его границы изображены выходящими за пределы фрейма данной диаграммы. Сообщение с именем `вводПИНкода()` изображается от границы фрейма диаграммы, что соответствует рассматриваемому далее шлюзу данного взаимодействия.

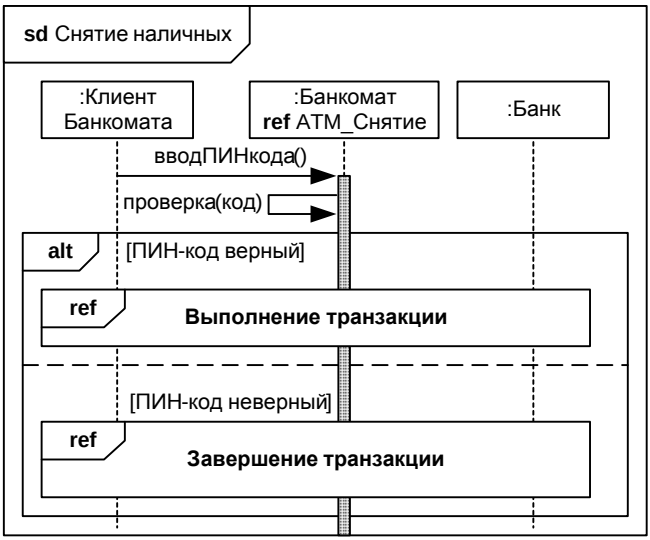


Рис. 7.12. Пример декомпозиции части в форме ссылки в заголовке линии жизни

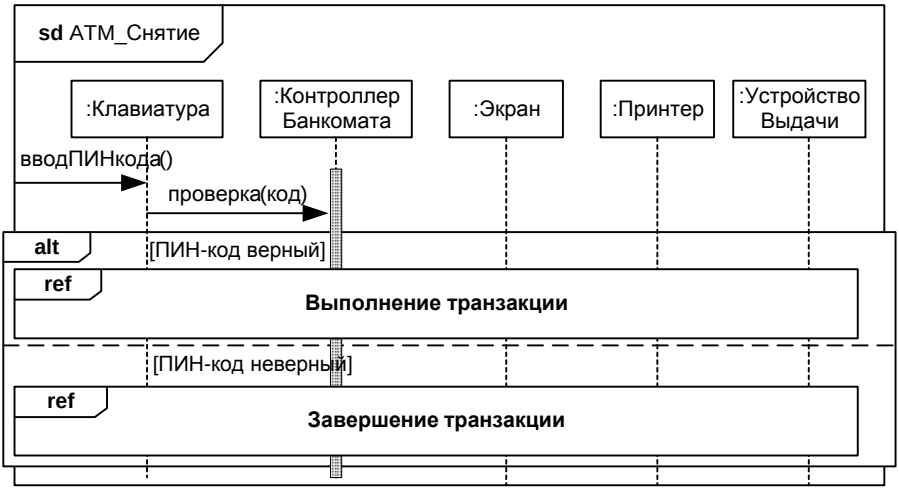


Рис. 7.13. Пример диаграммы последовательности для декомпозиции части

При использовании декомпозиций частей необходимо соблюдать следующее ограничение: декомпозиции частей применяются только к частям, которые являются частями внутренних структур, но не частями кооперации. Если на диаграмме последовательности имеются комбинированные фрагменты, которые покрывают все линии жизни, включая декомпозицию части, то такие

фрагменты являются глобальными. Диаграмма последовательности декомпозиции части должна включать все глобальные фрагменты исходного взаимодействия.

Декомпозиция части может быть изображена с помощью вложенных фреймов и дополнительным оператором строгого следования **strict**. Это означает, что данные фреймы на всех внутренних линиях жизни декомпозиции являются строго упорядоченными во времени относительно их вертикальной позиции на диаграмме последовательности.

7.5.3. Инвариант состояния

Инвариант состояния (state invariant) является некоторым ограничением времени выполнения, которое должно быть выполнено для отдельных участников взаимодействия.

Инвариант состояния может быть использован для спецификации различных видов ограничений, накладываемых на значения атрибутов или переменных, внутренние или внешние состояния и т. д. Инвариант состояния на диаграмме последовательности является фрагментом взаимодействия и размещается на линии жизни.

Инвариант состояния изображается в форме символа состояния на линии жизни соответствующего участника взаимодействия. В этом случае символ состояния представляет эквивалент ограничения, которое проверяет состояние объекта, представленного данной линией жизни. Это может быть внутреннее состояние поведения объекта соответствующего класса или некоторое внешнее состояние, основанное на представлении "черный ящик" для данной линии жизни.

В первом случае, если поведение класса описывается некоторым конечным автоматом, то имя этого состояния должно соответствовать имени соответствующего состояния конечного автомата (рис. 7.14). Так, например, состояние `Проверка ПИН-кода` должно присутствовать на диаграмме конечного автомата банкомата. Фрагмент *Альтернативы* будет выполнен только в том случае, если банкомат находится в этом состоянии. В противном случае траектории этого фрагмента рассматриваются как недействительные (недопустимые).

Во втором случае может быть указано только ограничение инварианта. Ограничение, ассоциированное с некоторой линией жизни, может быть изображено в форме текста в фигурных скобках на линии жизни (рис. 7.15). Это ограничение принимает значение "истина", если информация о специфицированном состоянии принимает значение "истина", и наоборот. Так, например, ограни-

чение {ПИН-код введен} специфицирует инвариант состояния для линии жизни класса Банкомат, которое должно принимать значение "истина" для выполнения последующего фрагмента *Альтернативы*.

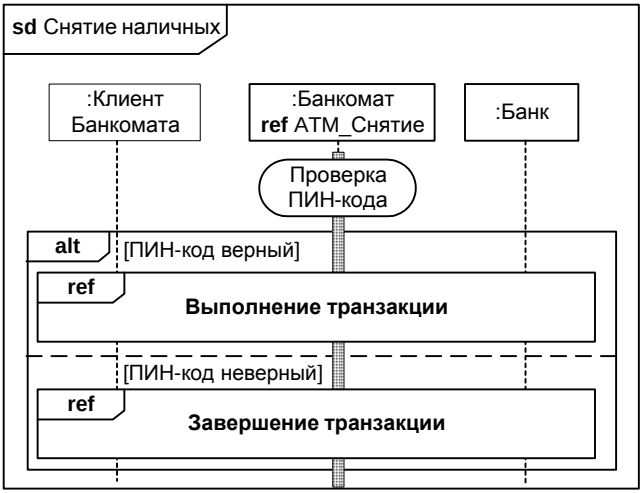


Рис. 7.14. Пример представления инварианта состояния в форме символа состояния

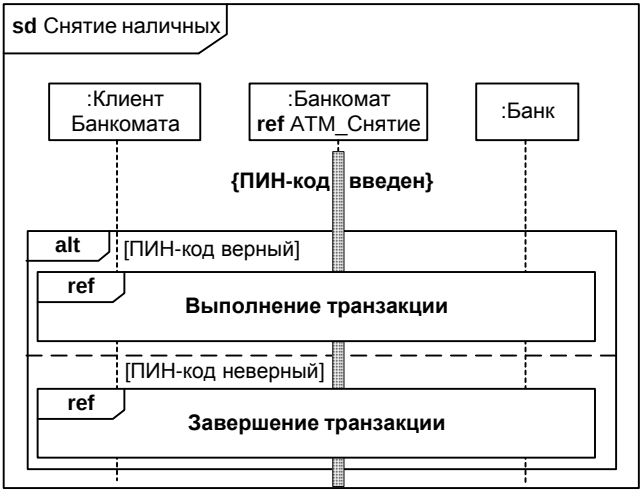


Рис. 7.15. Пример представления инварианта состояния в форме ограничения

Предполагается, что ограничение инварианта состояния должно оцениваться в ходе времени выполнения непосредственно перед выполнением следующей спецификации наступления события. При этом все действия, которые моделируются неявно, должны быть выполнены. Если ограничение инварианта принимает значение "истина", то данная траектория выполнения является допустимой траекторией. Если же ограничение инварианта принимает значение "ложь", то данная траектория является недопустимой траекторией. Другими словами, все последующие траектории, которые имеют инвариант состояния со значением ограничения "ложь", рассматриваются как недопустимые.

Инвариант состояния может быть также изображен в форме примечания, ассоциированного с некоторой спецификацией наступления события.

7.5.4. Продолжение

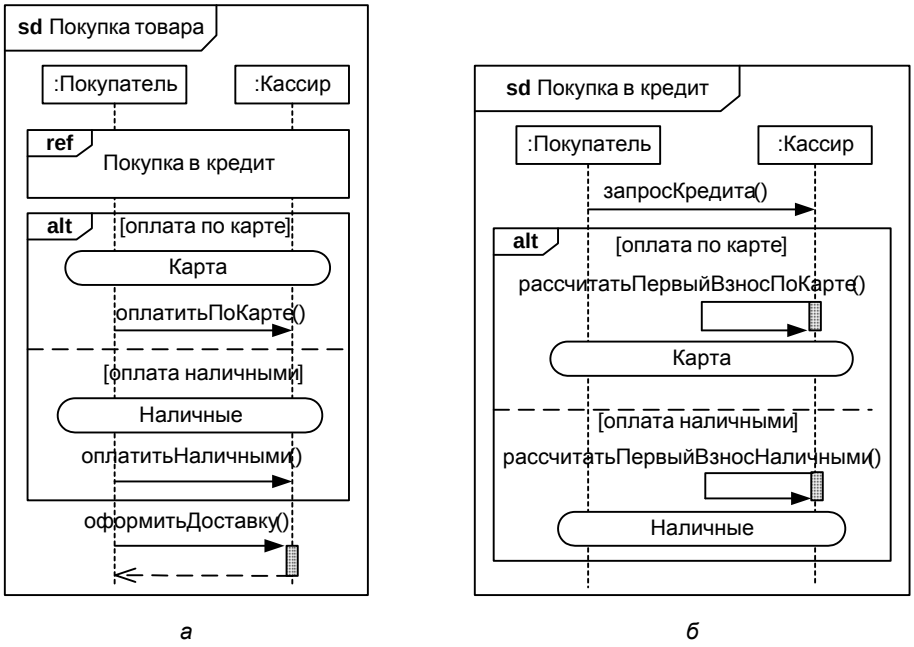
Продолжение (continuation) представляет собой некоторую метку, которая позволяет разбивать операнды комбинированного фрагмента *Альтернативы* на две и более части и комбинировать их траектории в различных фреймах.

Метки продолжения интуитивно представляют промежуточные точки в потоке управления комбинированного фрагмента *Альтернативы* и могут находиться в начале или конце этого фрагмента. Продолжение имеет смысл только в контексте комбинированного фрагмента *Альтернативы* и слабого следования. Если операнд взаимодействия комбинированного фрагмента *Альтернативы* заканчивается некоторым продолжением, то к этому операнду могут быть добавлены только фрагменты взаимодействия, начиная с этого продолжения или вовсе без этого продолжения.

Продолжение изображается символом состояния, но этот символ, в отличие от инварианта состояния, может покрывать более чем одну линию жизни. Продолжения, которые являются единственными в некотором комбинированном фрагменте, рассматриваются в конце этого фрагмента взаимодействия.

В качестве примера рассмотрим два фрагмента диаграммы последовательности при выполнении покупок в магазине (рис. 7.16). На первой диаграмме представлены два продолжения с метками `Карта` и `Наличные`, которые используются в операндах комбинированного фрагмента *Альтернативы* (рис. 7.16, *а*). При этом фрейм `Покупка в кредит` рассматривается как использование взаимодействия и представлен на отдельной диаграмме (рис. 7.16, *б*).

Если при покупке в кредит величина первого взноса отличается в зависимости от формы оплаты, то это может быть изображено в форме ссылок на продолжения в соответствующих операндах фрагмента *Альтернативы*.



а *б*

Рис. 7.16. Пример представления продолжений на диаграммах последовательности: *а* — на исходной диаграмме; *б* — на диаграмме использования взаимодействия

Полное множество траекторий двух представленных взаимодействий определяется как объединение соответствующих траекторий, при этом метки продолжения используются в качестве промежуточных точек общего потока управления. Тем самым две рассмотренные диаграммы эквивалентны в смысле множества их траекторий следующей диаграмме последовательности (рис. 7.17).

При использовании в модели продолжений следует соблюдать следующие ограничения. Продолжения с одним и тем же именем могут покрывать на разных диаграммах только одно и то же множество линий жизни. Продолжения являются всегда глобальными в фрагменте взаимодействия, который их содержит. Продолжение может быть или первым, или последним фрагментом взаимодействия в соответствующих операндах.

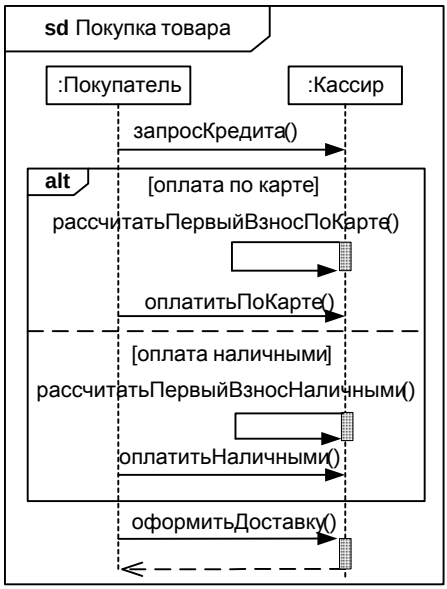


Рис. 7.17. Интерпретация продолжений на одной диаграмме последовательности

Что касается рассмотренного примера (см. рис. 7.16 и 7.17), то ситуация с покупкой в кредит может быть представлена в форме дополнительного сервиса, от которого, вообще говоря, покупатель может и отказаться. Модифицировать данную диаграмму последовательности с учетом этого замечания предлагается читателям самостоятельно в качестве упражнения.

Продолжения могут также представляться на линиях потоков управления диаграмм обзора взаимодействия, которые рассматриваются далее в *главе 9*.

7.5.5. Шлюз

Шлюз (gate) является точкой соединения для установления связи сообщения за пределами фрагмента взаимодействия с сообщением внутри этого фрагмента взаимодействия.

Шлюз предназначен для представления спецификации наступления события, которое происходит за пределами контекста рассматриваемого взаимодействия. Цель шлюзов как точек взаимодействия состоит в том, чтобы определить источник и цель сообщений или отношение порядка их следования. Шлюзы могут играть во взаимодействии различные роли, например, могут быть формальные шлюзы во фреймах взаимодействия, действительные шлюзы

в использовании взаимодействия и шлюзы выражений в комбинированных фрагментах.

Шлюзы изображаются в форме неявных точек на фрейме, которые соответствуют концам сообщений. Один и тот же шлюз может представляться несколько раз на одной и той же или различных диаграммах. Шлюзы могут иметь явное имя, чтобы иметь возможность ссылаться на них в различных диаграммах. Например, сообщение с именем `вводПИН-кода()` (см. рис. 7.13) имеет в качестве источника некоторый анонимный шлюз, изображенный на границе фрейма соответствующей диаграммы. Это дает возможность формального установления порядка следования сообщений во вложенных фрагментах взаимодействия.

Другими словами, шлюзы фрагментов взаимодействия представляют синтаксический интерфейс между фрагментом и его окружением, через который могут передаваться сообщения извне в рассматриваемый фрагмент, а также и вовне этого фрагмента. В последнем случае источник сообщения находится внутри рассматриваемого фрагмента.

При использовании в модели шлюзов следует соблюдать такие ограничения. Сообщение, направленное в (из) действительный шлюз некоторого использования взаимодействия, должно соответствовать сообщению, направленному из (в) формального шлюза с тем же именем взаимодействия, на которое ссылается рассматриваемое использование взаимодействия. Сообщение, направленное в (из) шлюз некоторого комбинированного фрагмента, должно соответствовать сообщению, направленному из (в) этого комбинированного фрагмента за его пределы.

Шлюзы могут быть идентифицированы или по имени, если оно явно указано на диаграмме, или неявно по некоторому заданному идентификатору. В последнем случае имя шлюза может быть образовано конкатенацией направления сообщения и имени сообщения, например, `in_вводПИН-кода()`, `out_вернутьКарту()`.

7.6. Специальные ограничения на диаграммах последовательности

Передача сообщений и выполнение операций на диаграмме последовательности может потребовать явной спецификации ограничений, накладываемых на интервал выполнения операций или время передачи сообщений. Ограничения на диаграмме последовательности могут относиться как к выполнению операций линиями жизни, так и к передаче сообщений, явно специфицируя требования по времени их передачи или приема.

Напомним, что ограничения в языке UML 2.0 всегда записываются в фигурных скобках. В отличие от условий ветвления, которые могут выполняться или не выполняться, временные и другие типы ограничений имеют обязательный или директивный характер для ассоциированных с ними элементов модели.

В общем случае для записи ограничений, которые являются одним из трех механизмов расширения элементов языка UML, может быть использован естественный язык, псевдокод или язык объектных ограничений (OCL). В данном разделе рассматриваются особенности спецификации временных ограничений на диаграммах последовательности.

7.6.1. Временное выражение

Временное выражение (time expression) определяет спецификацию некоторого значения времени.

Временное выражение обозначает некоторое значение времени и ссылается на некоторую спецификацию наступления события. Временное выражение является значением абсолютного времени, заданного в текстовом формате, форма которого часто зависит от реализации.

Поскольку время на диаграмме последовательности представляется в форме дискретных моментов наступления событий, временное выражение является неотрицательным целочисленным выражением. Это выражение специфицирует количество "щелчков времени", прошедших после некоторой заданной начальной точки.

При использовании временного выражения неявно используется атрибут с именем `firstTime`, который принимает булево значение. Если значением этого атрибута является "истина", то временное выражение описывает первую точку во времени наступления события для соответствующего элемента. Если значением этого атрибута является "ложь", то временное выражение описывает последнюю точку во времени наступления события для соответствующего элемента.

7.6.2. Временное событие

Временное событие (time event) специфицирует некоторый момент времени посредством выражения.

Временное событие специфицирует точку во времени посредством некоторого выражения. Выражение может соответствовать абсолютному или относи-

тельному времени. Для записи временного события используется следующий формальный синтаксис (БНФ):

<*временное-событие*>::= 'after' <*выражение*> | 'at' <*выражение*>

где 'after' <*выражение*> соответствует спецификации относительного временного события, 'at' <*выражение*> соответствует спецификации абсолютного временного события.

Другими словами, триггер относительного времени специфицируется посредством ключевого слова after, после которого следует некоторое временное выражение, заключенное обычно в круглые скобки, например, after (5 секунд). При этом начальной точкой события является момент времени, в который триггер становится активным.

Триггер абсолютного времени специфицируется с ключевым словом at, после которого следует временное выражение, заключенное обычно в круглые скобки, например, at (1 января 2007 г.).

На практике может иметь место некоторая задержка между временем приема и временем диспетчеризации некоторого временного события, например, по причине задержек в очереди. Моделирование этой ситуации в языке UML 2.0 является точкой изменения семантики и требует от разработчиков дополнительной спецификации модели взаимодействия.

7.6.3. Действие наблюдения времени

Действие наблюдения времени (time observation action) определяется как действие, которое наблюдает текущую точку во времени и записывает это значение в некоторую структурную характеристику.

В общем случае действие наблюдения времени специфицируется в форме назначения некоторого временного выражения структурной характеристике или атрибуту. При выполнении действия наблюдения времени оно получает текущее значение времени и записывает это значение в заданную структурную характеристику. Для спецификации действия наблюдения времени используется следующий формальный синтаксис (БНФ):

<*действие-наблюдения-времени*>::= {<*имя-атрибута*> '= now'}

Спецификация действия наблюдения времени ассоциируется с соответствующим элементом модели посредством линии. Например, в рассматриваемой системе контроля (рис. 7.18) сообщение о срабатывании датчика ассоциировано с действием наблюдения времени, которое фиксирует время момента срабатывания и записывает это значение в качестве значения атри-

бута t. Далее это значение может использоваться для спецификации временных ограничений.

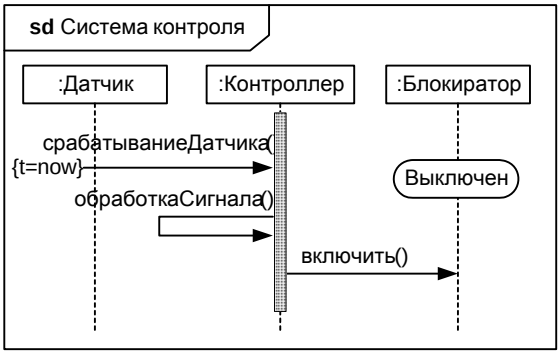


Рис. 7.18. Наблюдение времени на диаграмме последовательности

Для записи временных ограничений необходимо использовать понятие временного интервала.

7.6.4. Интервал

Интервал (interval) определяет диапазон между двумя спецификациями значений.

Семантика интервала всегда относится к ограничениям, в которых он принимает участие. Интервал обозначается текстом посредством двух спецификаций значений, разделенных символом двоеточия `..`:

`<`*интервал*`>::= <`*min-значение*`> '..' <`*max-значение*`>`

где `<`*min-значение*`>` — минимальное значение диапазона спецификации значений, `<`*max-значение*`>` — максимальное значение диапазона спецификации значений.

Частными случаями интервала являются временной интервал и интервал продолжительности

Временной интервал (time interval) определяет диапазон между двумя временными выражениями.

Временной интервал изображается в форме интервала, где каждый элемент спецификации значения является временным выражением.

7.6.5. Временное ограничение

Временное ограничение (time constraint) представляет собой специальное ограничение, записанное в форме временного интервала.

Все последующие траектории взаимодействия, для которых временные ограничения не выполняются, являются отрицательными или недопустимыми траекториями. Это означает, что если они происходят на практике, то система или выходит из строя, или не удовлетворяет базовым требованиям.

Временное ограничение изображается в форме графической ассоциации между временным интервалом и конструкцией, которую оно ограничивает. Обычно эта графическая ассоциация является небольшой линией, например, между наступлением события передачи или приема сообщения и временным интервалом.

Продолжая рассмотрение примера фрагмента диаграммы последовательности для системы контроля (рис. 7.19), дополним событие передачи сообщения о включении блокиратора временным ограничением `{t..t+2 sec}`. Это означает, что интервал времени между срабатыванием датчика и включением блокиратора не должен превышать 2-х секунд.

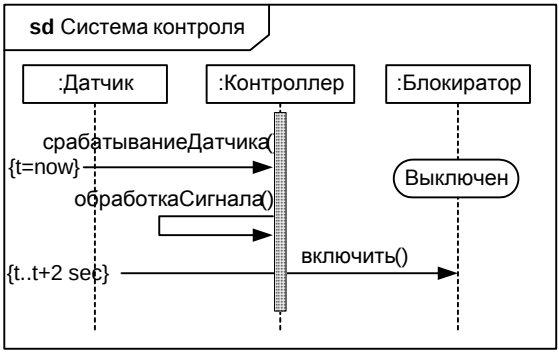


Рис. 7.19. Временное ограничение

Кроме временных ограничений на диаграммах последовательности могут использоваться также ограничения на продолжительность.

7.6.6. Продолжительность

Продолжительность (duration) специфицирует временное расстояние между двумя временными выражениями, которые соответствуют двум моментам времени.

В общем случае продолжительность ассоциируется с событиями двух элементов модели. В этом случае неявно существуют два логических атрибута, один для записи начала продолжительности и один для записи конца продолжительности.

Продолжительность определяет спецификацию значений, которая обозначает некоторую продолжительность во времени. Продолжительность задается разностью во времени между начальной точкой во времени и конечной точкой во времени. Если конечная точка во времени предшествует начальной точке во времени, то продолжительность, тем не менее, будет положительной, предполагая, что начальная и конечная точки должны поменяться местами.

Продолжительность является значением относительного времени, заданного в произвольном текстовом формате. Как правило, продолжительность специфицируется в форме неотрицательного целочисленного выражения, представляющего количество "щелчков времени", которые могут произойти в ходе этой продолжительности. Продолжительность может быть записана также текстом на обычном языке и содержать арифметические операторы.

Для записи ограничений на продолжительность используется понятие интервала продолжительности.

Интервал продолжительности (duration interval) определяет диапазон между двумя продолжительностями.

Интервал продолжительности изображается с использованием нотации интервала, где каждый элемент спецификации значения является выражением продолжительности.

7.6.7. Действие наблюдения продолжительности

Действие наблюдения продолжительности (duration observation action) определяется как действие, которое наблюдает продолжительность во времени и записывает это значение в некоторую структурную характеристику.

При выполнении действия наблюдения продолжительности оно измеряет идентифицируемую продолжительность и записывает полученное значение в заданную структурную характеристику. Для спецификации действия наблюдения продолжительности используется следующий формальный синтаксис (БНФ):

<*действие-наблюдение-продолжительности*>::= <*имя-атрибута*>'=duration'

Наблюдение продолжительности для наглядности может ассоциироваться с соответствующими элементами модели линиями.

В качестве примера использования наблюдения продолжительности рассмотрим фрагмент диаграммы последовательности по обработке запросов к БД (рис. 7.20). Пользователь БД передает сообщение с именем `запрос(код)`, в ходе передачи которого измеряется его продолжительность, т. е. время между передачей и приемом этого сообщения. Значение этой измеренной продолжительности присваивается атрибуту с именем `d`. Далее СУБД выполняет обработку запроса, в ходе выполнения которого также измеряется его продолжительность. Во втором случае наблюдение продолжительности записывает продолжительность выполнения сообщения в качестве значения атрибуту с именем `d'`.

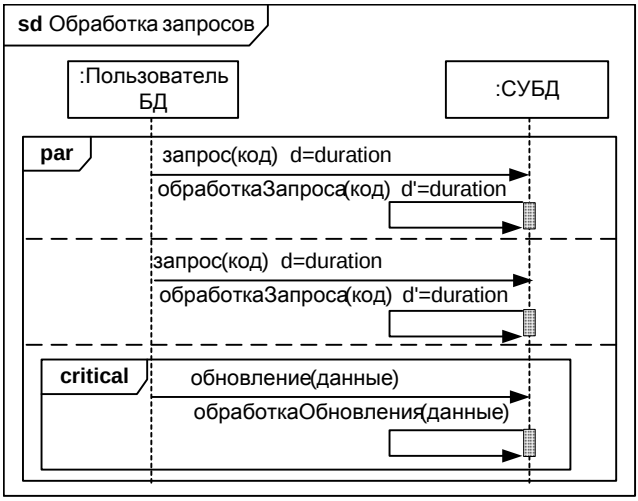


Рис. 7.20. Наблюдение продолжительности

Наблюдение продолжительности используется для формирования ограничений на продолжительность.

7.6.8. Ограничение на продолжительность

Ограничение на продолжительность (duration constraint) определяет ограничение, которое ссылается на некоторый интервал продолжительности.

Ограничение на продолжительность изображается в форме некоторой графической ассоциации между интервалом продолжительности и конструкциями,

которые оно ограничивает. Эта ассоциация может быть явной или неявной. Явная форма предполагает использование линий для указания элементов модели, к которым относится ограничение на продолжительность. Неявная форма изображает ограничение на продолжительность вблизи соответствующих элементов модели, в качестве которых могут быть наступления событий передачи и приема сообщений.

В качестве примера дополним фрагмент диаграммы последовательности по обработке запросов к БД (см. рис. 7.20) ограничениями на продолжительность (рис. 7.21). Первое ограничение на продолжительность ассоциируется с сообщением `запрос(код)`, продолжительность передачи которого не должно превышать 10 миллисекунд. Второе ограничение на продолжительность ассоциируется с обработкой запроса, продолжительность выполнения которого не должно превышать 1 секунды.

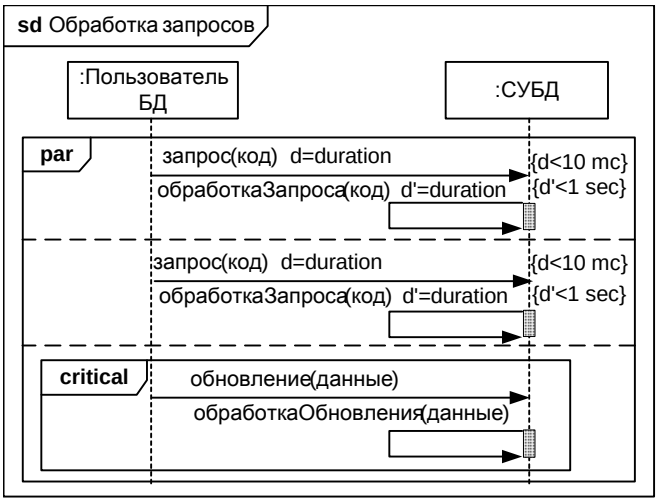


Рис. 7.21. Ограничение на продолжительность

Рассмотренный пример модели взаимодействия по обработке запросов к БД может быть дополнен с целью учета реальных требований к разрабатываемой системе. Например, можно учесть требование максимального количества одновременно обрабатываемых запросов, возможность постановки запросов в очередь, время ожидания запросов в очереди и пр. Модифицировать данную диаграмму последовательности с учетом этих дополнительных требований предлагается читателям самостоятельно в качестве упражнения.

* * *

В заключение приводится сводка основных графических обозначений, которые могут быть использованы при построении диаграмм последовательности (табл. 7.1).

Таблица 7.1. *Графические узлы и пути на диаграммах последовательности*

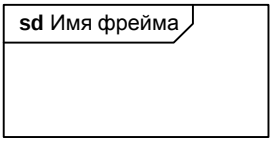
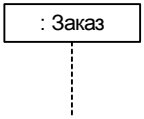
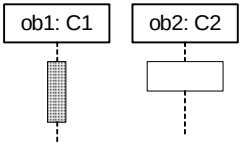
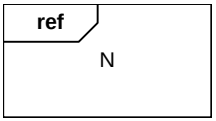
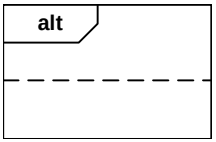
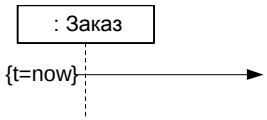
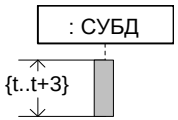
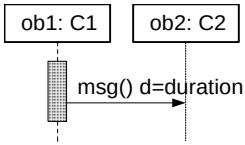
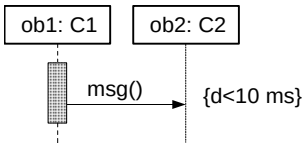
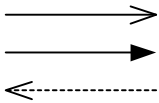
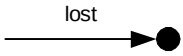
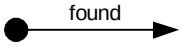
Тип графического элемента	**Нотация**	**Ссылка на раздел**
Фрейм (frame)		Диаграмма последовательности *(см. разд. 7.1)*
Линия жизни (lifeline)		Линия жизни *(см. разд. 7.2)*
Спецификация выполнения (execution specification)		Линия жизни *(см. разд. 7.2)*
Использование взаимодействия (interaction use)		Использование взаимодействия *(см. разд. 7.5.1)*
Комбинированный фрагмент (combined fragment)		Комбинированный фрагмент *(см. разд. 7.4)*
Инвариант состояния (state invariant) и Продолжение (continuation)		Инвариант состояния *(см. разд. 7.5.3)*; продолжение *(см. разд. 7.5.4)*
Уничтожение (destroy)		Линия жизни *(см. разд. 7.2)*

Таблица 7.1 *(окончание)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Наблюдение времени (time observation)		Действие наблюдения времени *(см. разд. 7.6.3)*
Временное ограничение (time constraint)		Временное ограничение *(см. разд. 7.6.5)*
Наблюдение продолжительности (duration observation)		Действие наблюдения продолжительности *(см. разд. 7.6.7)*
Ограничение на продолжительность (duration constraint)		Ограничение на продолжительность *(см. разд. 7.6.8)*
Сообщение (message). Здесь изображены: асинхронное сообщение, вызов операции и ответ		Сообщение *(см. разд. 7.3.1)*
Потерянное сообщение (lost message)		Сообщение *(см. разд. 7.3.1)*
Найденное сообщение (found message)		Сообщение *(см. разд. 7.3.1)*

Глава 8

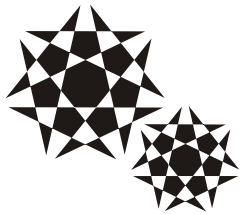


Диаграмма деятельности (activity diagram)

В общем случае поведение может рассматриваться в нескольких аспектах, следствием чего является наличие нескольких типов диаграмм для моделирования поведения в языке UML 2.0. Во-первых, поведение может находиться в собственности классификаторов и описывать изменение состояний отдельных объектов с течением времени. Для моделирования такого поведения используются диаграммы конечного автомата, которые описываются в *главе 10*. Во-вторых, поведение может относиться к взаимодействию объектов или линий жизни при реализации функциональности системы. Для моделирования такого поведения могут применяться различные диаграммы взаимодействия, которые обсуждаются в *главах 7* и *9*. Наконец, поведение может характеризоваться некоторой деятельностью в форме последовательности действий, которые выполняются различными элементами, входящими в состав системы. Для моделирования такого поведения используются диаграммы деятельности, которые служат предметом рассмотрения в данной главе.

Центральным аспектом моделирования деятельности в языке UML 2.0 является последовательность действий и условий их выполнения, а также поток объектов, которые являются необходимыми для выполнения отдельных действий или являются результатом их выполнения. Поэтому диаграмма деятельности в языке UML 2.0 содержит целый ряд новых понятий, предназначенных для детальной спецификации потока управления и потока объектов.

8.1. Концептуальные основы моделирования деятельности

Деятельность в языке UML 2.0 уже не является специализацией состояния и не использует понятие графа деятельности, как это было в UML 1.*x*. Вместо семантики конечных автоматов для моделирования деятельности используется

механизм, аналогичный сетям Петри. Это позволяет рассматривать не только поток управления, но также потоки объектов и данных. На диаграммы деятельности UML 2.0 также оказала влияние нотация моделирования бизнес-процессов (BPMN), что привело к горизонтальному расположению потока управления и потока объектов на этих диаграммах.

В общем случае деятельность является самостоятельным элементом поведения, которая, в свою очередь, может включать в себя другие деятельности или отдельные действия. При этом элементарной единицей спецификации поведения является действие. Последовательность выполнения действий в деятельности описывается с помощью потока управления, который лежит в основе понимания модели поведения в представлении диаграммы деятельности.

8.1.1. Деятельность и действие

Деятельность (activity) является спецификацией параметризованного поведения в форме координируемой последовательности подчиненных единиц, индивидуальными элементами которых являются действия.

Для спецификации выполнения подчиненного поведения деятельности служит некоторая модель потока управления и потока данных. При этом поток управления моделируется в форме узлов деятельности, соединенными дугами деятельности. Кроме узлов деятельности на диаграмме деятельности используются специальные конструкции для управления, такие как синхронизация, решение и управление параллельностью. Деятельности могут образовывать иерархии, в которых одни деятельности вызывают другие.

Элементами, из которых состоят деятельности, являются действия.

Действие (action) представляет собой элементарную единицу спецификации поведения, которая не может быть далее декомпозирована в форме деятельности.

Действие является именованным элементом, которому соответствует один шаг или этап при выполнении некоторой деятельности. Выполнение действия представляет собой некоторое преобразование или обработку в моделируемой системе, например, выполнение вычислений, запросов к базе данных или чего-либо еще. При выполнении действия оно принимает множество входов и преобразует их в множество выходов, при этом одно или оба эти множества могут быть пустыми. В результате выполнения действия может измениться состояние системы. Каждое действие в деятельности может выполняться произвольное количество раз.

Значения, которые являются входами действия, могут быть представлены в форме спецификаций, которые описывают необходимые входы для действий из выходов других действий. Поведение, к которому относятся действия, может специфицировать дополнительные ограничения, с целью уточнения характера выполнения отдельных действий и требований относительно их входов и выходов. Действия могут иметь доступ к данным с целью их преобразования или тестирования.

Действия не могут быть декомпозированы в деятельности, которая их содержит. Однако выполнение некоторого действия может индуцировать выполнение других действий. Например, действие может вызывать операцию, которая, в свою очередь, реализуется деятельностью и содержит действия. Примерами других элементарных действий являются передача сигнала и вызов поведения.

Концепция деятельности языка UML 2.0 может быть применена к моделированию бизнес-процессов и потоков работ. В этом случае события, которые управляют выполнением действий, могут порождаться как внутри системы, например, окончание выполнения задач, так и вне системы, например, вызов клиента. Деятельности могут также использоваться при моделировании информационных систем для спецификации процессов системного уровня.

8.1.2. Узлы и дуги деятельности

Основными графическими элементами на диаграмме деятельности являются различные узлы и дуги деятельности.

Узел деятельности (activity node) является абстрактным классом для отдельных точек в потоке деятельности, соединенных дугами.

Узлы деятельности могут относиться только к отдельной деятельности или группе деятельностей. Существует несколько видов конкретных узлов деятельности, для каждого из которых имеется собственная графическая нотация: узел действия (рис. 8.1, *а*), узел вызова деятельности (рис. 8.1, *б*), узел объекта (рис. 8.1, *в*) и узел управления (рис. 8.1, *г*).

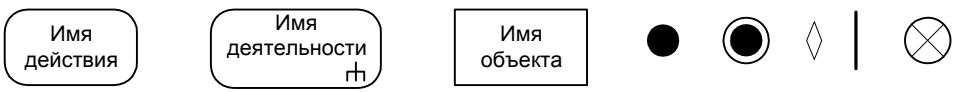


Рис. 8.1. Нотация для различных узлов деятельности: *а* — узел действия; *б* — узел вызова деятельности; *в* — узел объекта; *г* — узел управления

Узел деятельности может иметь имя, которое записывается с прописной буквы и в пределах одной диаграммы может повторяться с целью моделирования нескольких вызовов одного и того же поведения или многократного выполнения одного и того же действия.

Узел вызова деятельности изображается размещением символа "грабли" в символ узла деятельности. Эти "грабли" имеют сходство с символом иерархии, призванным иллюстрировать тот факт, что этот вызов деятельности начинает другую деятельность. Последняя может быть представлена в форме дополнительной вложенной диаграммы деятельности. Более подробно отдельные виды узлов действий, объектов и управления рассматриваются далее в этой главе.

На диаграмме узлы деятельности могут быть связаны между собой при помощи дуг деятельности.

Дуга деятельности (activity edge) является абстрактным классом для направленных соединений между двумя узлами деятельности.

Дуги деятельности могут находиться в собственности только деятельности или групп деятельностей. Дуга деятельности является направленным отношением и имеет в качестве источника и цели некоторые узлы. При этом источник и цель дуги должны относиться к той же деятельности, что и сама дуга.

Дуга деятельности изображается сплошной линией с тонкой стрелкой, при этом имя дуги может отсутствовать (рис. 8.2, *а*). Если же дуга имеет имя, то оно записывается со строчной буквы и указывается вблизи стрелки (рис. 8.2, *б*).

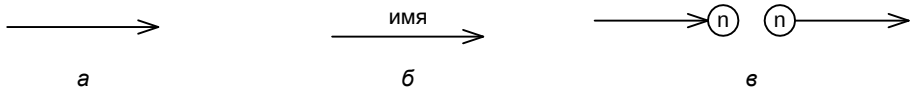


Рис. 8.2. Нотация для различных дуг деятельности: *а* — имя дуги отсутствует; *б* — дуга имеет имя; *в* — дуга с соединителем

Дуга деятельности может быть также изображена с использованием соединителя, который представляется в форме небольшого круга с именем метки для этой дуги (рис. 8.2, *в*). Это обозначение принято исключительно с целью удобства и не оказывает влияния на модель. При этом каждый такой соединитель с заданной меткой должен быть в точности равен другому соединителю с той же самой меткой на этой же диаграмме деятельности. Один из каждой пары соединителей должен иметь только одну входящую дугу, а другой — только одну выходящую дугу.

Для дуги деятельности рядом с линией может быть изображен вес дуги в фигурных скобках. Вес является спецификацией количественного значения, которое может быть равно положительному целому (рис. 8.3, *а*) или `null` (рис. 8.3, *б*). При этом значение `null` обозначает "все". Если регионы деятельности имеют прерывания, то для соответствующей дуги деятельности, которое изображает это прерывание, используется специальная нотация — линия в стиле стрелки молнии (рис. 8.3, *в*).

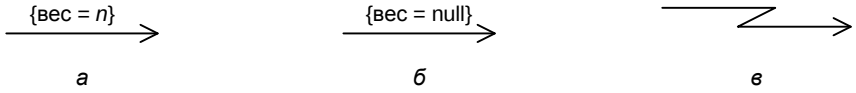


Рис. 8.3. Нотация для дуги: *а*, *б* — с весом; *в* — для прерываемых регионов

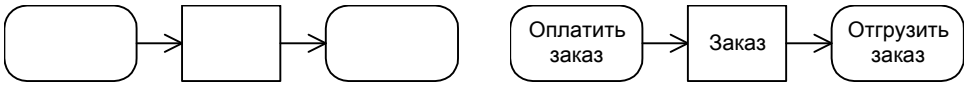
Существуют две разновидности дуг деятельности: дуги потока управления и дуги потока данных. Они обозначаются одинаково — линией с тонкой стрелкой. *Поток управления* (control flow) представляется в форме дуги деятельности, которая связывает между собой два узла деятельности (рис. 8.4, *а*) и по которой передаются только маркеры управления. При этом все маркеры управления, которые предлагаются узлом-источником, предлагаются узлу цели, а вдоль дуги потока управления не могут следовать объекты и данные.

В качестве примера можно рассмотреть фрагмент передачи управления от действия `Оплатить заказ` к действию `Отгрузить заказ`. Дуга деятельности между ними является потоком управления, который указывает на то, что когда закончится выполнение действия `Оплатить заказ`, то будет вызвано действие `Отгрузить Заказ` (рис. 8.4, *б*).

Рис. 8.4. Дуги потока управления: *а* — способ представления; *б* — пример

Поток объектов (object flow) представляется в форме дуги деятельности, по которой передаются только маркеры объектов или данных (рис. 8.5, *а*). При этом все маркеры, предлагаемые узлом-источником, предлагаются для узла цели с учетом ограничений, которые могут быть дополнительно специфицированы с помощью веса дуги. Узлы объектов, соединенные потоком объектов с необязательными промежуточными узлами действий или управления,

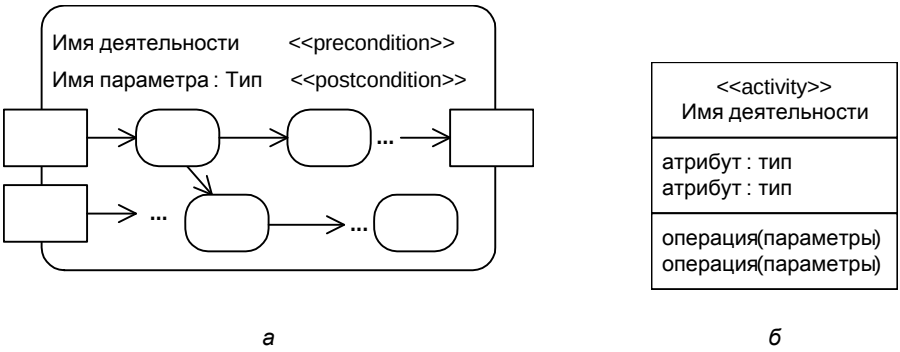
должны иметь совместимые типы. В качестве примера можно рассмотреть фрагмент передачи объектов `Заказ` от действия `Оплатить заказ` к действию `Отгрузить заказ`. Здесь обе линии со стрелками являются дугами потока объектов (рис. 8.5, *б*).



а *б*

Рис. 8.5. Дуги потока объектов: *а* — способ представления; *б* — пример

Два потока объектов могут иметь один и тот же узел объекта в качестве источника. В этом случае дуги будут конкурировать за объекты. Если одна дуга приняла маркер объекта из узла объекта, другие дуги уже не будут иметь к нему доступа. Потоки объектов поддерживают многократную передачу и прием маркеров, выбор маркеров из узлов объектов и преобразование маркеров. Однако для многократного использования маркеров следует применять узел разделения. Дополнительные особенности и варианты изображения потока объектов рассматриваются далее в этой главе.



а *б*

Рис. 8.6. Варианты нотации для деятельности: *а* — способ представления; *б* — деятельность является классом

Деятельность изображается в форме прямоугольника с закругленными вершинами, который образует границу этой деятельности. Внутри этого символа размещаются узлы и дуги деятельности, а также указывается ее имя в верхнем левом углу. Дополнительно могут быть изображены ограничения на предусловия и постусловия деятельности с ключевыми словами `<<precondition>>` и `<<postcondition>>` соответственно, которые имеют глобальный характер.

Узлы параметров деятельности изображаются на границе символа (рис. 8.6, *а*). Для изображения на диаграмме только характеристик деятельности также может быть использована нотация для классов с ключевым словом `<<activity>>`. Как правило, этот вариант используется с целью показать, что деятельность является классом (рис. 8.6, *б*).

Прямоугольник с закругленными углами для изображения деятельности может быть заменен нотацией фрейма аналогично изображению фрейма диаграммы. В этом случае узлы параметров деятельности изображаются на фрейме. В отдельных случаях символ деятельности или фрейм могут быть опущены вовсе.

8.1.3. Семантика деятельности

Семантика деятельности в языке UML 2.0 основывается на потоке маркеров.

Маркер (token) — элемент модели, предназначенный для представления некоторого объекта, данных или управления и существующий на диаграмме деятельности в отдельном узле.

Каждый маркер отличается от любого другого, даже если он содержит то же значение, что и другой. Любой узел деятельности может начать свое выполнение, только если удовлетворены специфицированные условия для его входных маркеров, причем эти условия зависят от вида узла. Когда узел начинает свое выполнение, маркеры принимаются из некоторых или всех его входных дуг, а специальный маркер размещается в этом узле. Когда узел завершает выполнение, специальный маркер удаляется из этого узла, а другие маркеры предлагаются в некоторых или всех его выходных дугах.

Маркеры могут протекать вдоль дуги, правила следования которых зависят от вида дуги и характеристик ее источника и цели. Маркер перемещается по дуге, если он одновременно удовлетворяет правилам для узла цели, дуги и узла источника. Это означает, что узел-источник может только предоставлять маркеры в выходящие дуги, а не направлять их дальше вдоль этих дуг, поскольку маркеры могут быть отвергнуты дугой или узлом целью на другой стороне.

В общем случае вдоль дуги может одновременно следовать любое количество маркеров, индивидуально или в группах, в одно или в разное время. Если указано значение веса дуги, то оно определяет минимальное количество маркеров, которые могут проследовать по дуге за один раз. Если источник предлагает некоторое количество маркеров, то все эти маркеры предлагаются для

цели одновременно. Если количество маркеров, которое источник предлагает цели, меньше веса дуги, то все маркеры совсем не предлагаются. Неограниченный вес означает, что все маркеры в источнике могут быть предложены цели. Несколько маркеров, предлагаемых некоторой дуге одновременно, является такими же, как если бы они предлагались по одному в некоторый момент времени.

Деятельность может иметь несколько маркеров, которые втекают в нее или вытекают из нее в любой момент времени. Для приема и хранения таких маркеров предназначены узлы объекта, действующие как буферы. В узлах объекта маркеры могут также ожидать выполнения некоторых условий, чтобы двигаться дальше. Однако маркеры не могут "покоиться" в узлах управления, поскольку узлы управления действуют как переключатели трафика управляющих маркеров, когда они совершают свой путь между узлами объекта и узлами действиями. В узлах действий маркеры могут оставаться в течение некоторого периода времени.

Во время выполнения деятельность имеет доступ к атрибутам и операциям объекта своего контекста и любых объектов, транзитивно связанных с объектом контекста. Деятельность, которая является методом характеристики поведения, имеет доступ к параметрам этой характеристики поведения и вызывается при вызове этой характеристики поведения.

Деятельности могут быть анонимными, если они не назначены никакому классификатору. Например, при реинжиниринге бизнес-процессов деятельность может быть оптимизирована независимо от того, каким департаментам или должностям будут назначены для выполнения ее действия. При этом маркеры деятельности могут достигать узких мест и ожидать прохождения других маркеров перед ними, чтобы самим двигаться дальше. Маркеры могут настигать друг друга во время выполнения вызываемого поведения и, что наиболее важно, могут прерывать деятельность с помощью специальной конструкции — окончания деятельности.

Одна деятельность может вызывать другую деятельность. Если для каждого вызова используется отдельное выполнение деятельности, маркеры от различных вызовов не взаимодействуют. При этом каждое новое выполнение деятельности для каждого вызова уменьшает взаимодействие маркеров, но не может полностью его устранить.

8.1.4. Семантика действия

Последовательность выполнения действий определяется дугами управления и дугами потоков объектов, которые несут маркеры управления и маркеры объектов соответственно. При этом действие может начать свое выполнение,

когда выполнены *предварительные условия* для его потоков управления и объектов: все входящие дуги имеют маркеры управления, а все входные контакты имеют маркеры объектов. Действие начинает выполнение изъятием маркеров из входящих в него дуг управления и входных контактов. Когда выполнение действия заканчивается, оно помещает маркеры в выходящие из него дуги управления и выходные контакты, где они становятся доступными для других действий.

Более детально правила выполнения действия могут быть представлены следующим образом.

- Выполнение действия становится возможным, когда удовлетворены предварительные условия для его потоков управления и объектов. Предварительное условие потока считается удовлетворенным, если все входные контакты имеют маркеры и предлагают их одновременно, предотвращая их поглощение любыми другими действиями. Это должно гарантировать, что другие действия, конкурирующие за маркеры, не смогут принять при своем многократном выполнении только часть этих маркеров. При этом исключается блокировка начала выполнения действия, когда оно будет ожидать маркеры, которые уже поглощены другими действиями.
- Выполнение действия поглощает входные маркеры управления и маркеры объектов и удаляет их из источников дуг управления и из входных контактов. Если на одной дуге являются доступными несколько маркеров управления, то они все поглощаются.
- Действие продолжает выполнение до тех пор, пока оно не будет завершено. На большинство действий можно оказывать влияние только через его входы. Однако некоторые действия могут предоставлять доступ к более широкому контексту, такому как переменные или даже к самому объекту, который владеет деятельностью, содержащей выполнение этого действия. Детальная семантика выполнения действия и определения завершения зависит от подкласса действия.
- После завершения действия оно предлагает маркеры объектов во все его выходные контакты, а маркеры управления во все выходящие из него дуги управления, и на этом формально оно заканчивается. С этого момента становятся доступными выходные маркеры, и может быть доступна проверка предварительных условий управления или потока объектов для выполнения других действий.
- После окончания выполнения действия с помощью некоторой реализации должны быть восстановлены его ресурсы. Это дает возможность повторного выполнения действия сразу после завершения его выполнения при удовлетворении соответствующих предварительных условий. Однако

конкретные особенности управления ресурсами не специфицированы в языке UML 2.0 и могут потребовать дополнительной спецификации.

Порядок выполнения двух или более действий в деятельности явно специфицируется отношением потока управления или объектов. Если два действия не являются непосредственно или транзитивно упорядоченными в каком-либо отношении потока, то потенциально они могут выполняться параллельно. Однако их параллельное выполнение вовсе не является необходимым, поскольку для представления параллельности следует использовать специальный узел управления.

В общем случае выполнение отдельного действия не может повторяться, поскольку в любой момент времени может существовать не более одного его выполнения. Другими словами, если действие выполняется, то повторный вызов этого действия не приведет к началу его повторного выполнения. В этом случае доступные маркеры управления не используются и могут собираться на входных контактах этого действия.

Если в ходе выполнения действия происходит некоторое исключение, то выполнение действия прекращается, и это действие не генерирует никакого выхода. Если действие имеет обработчик исключения, то он принимает маркер исключения в качестве объекта. При этом данные, описывающие исключение, представляются как объект любого класса.

Действие является отдельным дискретным элементом в спецификации деятельности. Однако оно может также вызывать другие деятельности по ссылке, что может привести к вызову других действий с произвольно сложным поведением. Это может привести к разработке несостоятельных диаграмм, в случае вызова рекурсивного поведения.

8.2. Узлы управления

Узел управления (control node) является абстрактным узлом деятельности, который предназначен для координации потоков в деятельности. Существует несколько разновидностей конкретных узлов управления: начальный узел, финальный узел и его прямые потомки, узел разделения, узел соединения, узел решения и узел слияния. Далее рассматриваются назначение и нотация каждого из этих узлов.

8.2.1. Начальный узел

Начальный узел (initial node) является узлом управления, в котором начинается поток при вызове деятельности.

Начальный узел является начальной точкой в выполнении деятельности. Когда деятельность вызывается, то в начальном узле размещается маркер управления, который предлагается всем выходящим из него дугам. Начальный узел не может иметь входящих дуг, а в качестве источника может иметь только дуги управления.

Начальный узел обозначается в форме круга. В качестве примера можно привести фрагмент начала некоторой деятельности, при вызове которой начальный узел отправляет управление в действие `Получить заказ` (рис. 8.7, *а*).

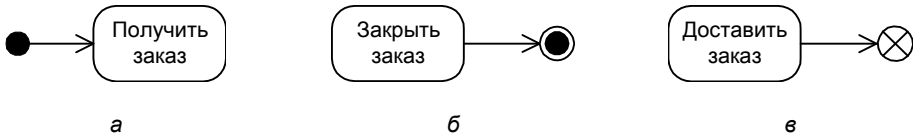


Рис. 8.7. Графические символы для изображения: *а* — начального узла; *б* — узла финала деятельности; *в* — финала потока

Деятельность может иметь несколько начальных узлов. Вызов такой деятельности приведет к началу нескольких потоков управления, каждому из которых будет соответствовать отдельный начальный узел.

Начальные узлы могут хранить маркеры управления, если они блокируются для дальнейшего использования последующими действиями, например, посредством сторожевых условий. Это эквивалентно размещению узла "центральный буфер" между начальным узлом и его выходящими дугами.

Следует заметить, что кроме начальных, потоки также могут начинаться и в других узлах, таких как узел параметра деятельности. Так что начальные узлы вовсе не являются необходимыми для деятельности, чтобы начать ее выполнение.

8.2.2. Узел финала деятельности и потока

Узел финала деятельности (activity final node) является узлом управления, который прекращает или останавливает все потоки в деятельности.

Узлы финала деятельности не имеют выходящих дуг, поэтому он только принимает все маркеры, которые предлагаются на входящих дугах. Маркер, достигающий финального узла деятельности, прекращает эту деятельность. В частности, он останавливает все выполняющиеся действия в этой деятель-

ности и разрушает все маркеры в узлах объектов, за исключением узлов деятельности с выходными параметрами.

Узел финала деятельности обозначается в форме окружности с вложенным в нее кругом. В качестве примера можно привести фрагмент окончания некоторой деятельности, когда после выполнения действия `Закрыть заказ` все маркеры в этой деятельности прекращаются (рис. 8.7, *б*). Это изображается передачей управления узлу финала деятельности.

Деятельность может иметь более чем один узел финала деятельности. При достижении первого из них прекращаются все потоки в этой деятельности, а сама она завершается.

Если по какой-либо причине нежелательно завершать все потоки в деятельности с несколькими узлами финала деятельности, то вместо них следует использовать узлы финала потока, которые рассматриваются далее.

Для завершения отдельного потока в деятельности предназначены узлы финала потока.

Узел финала потока (flow final node) является финальным узлом, который завершает отдельный поток управления или поток объектов, не завершая содержащей его деятельности.

Финал потока разрушает любые маркеры, которые достигают его. Он не оказывает влияния на другие потоки в этой деятельности. Узел финала потока обозначается в форме окружности с вложенным в нее значком "X". В качестве примера можно привести фрагмент окончания потока выполнения заявки, который прекращается после передачи управления узлу финала потока (рис. 8.7, *в*). При этом деятельность, которая содержит данный фрагмент, не прекращается и может выполняться в других потоках.

8.2.3. Узел решения

Узел решения (decision node) является узлом управления, который выбирает между выходящими потоками.

Узел решения должен иметь отдельную дугу деятельности, входящую в него, и одну или более дуг, выходящих из него. Дуги, входящие в узел решения и выходящие из него, должны быть все или потоками управления, или потоками объектов. Каждый маркер, достигающий узла решения, может перейти только в одну выходящую дугу. При этом маркеры не копируются.

Узел решения изображается символом ромба (рис. 8.8, *а*). Функциональность узла решения и рассматриваемого далее узла слияния может быть объединена с использованием одного символа ромба (рис. 8.8, *б*). Это эквивалентно изображению узла слияния со всеми изображенными входящими дугами и одной выходящей дугой, и узла решения, который имеет одну входящую и все изображенные выходящие дуги.

В качестве примера можно привести фрагмент диаграммы деятельности, на которой изображен узел решения, который следует за действием `Получить заказ`. Ветвление основывается на том, заказал ли клиент обычную или срочную доставку. В первом случае управление передается действию `Оформить обычную доставку`, а во втором случае — действию `Оформить срочную доставку` (рис. 8.8, *в*).

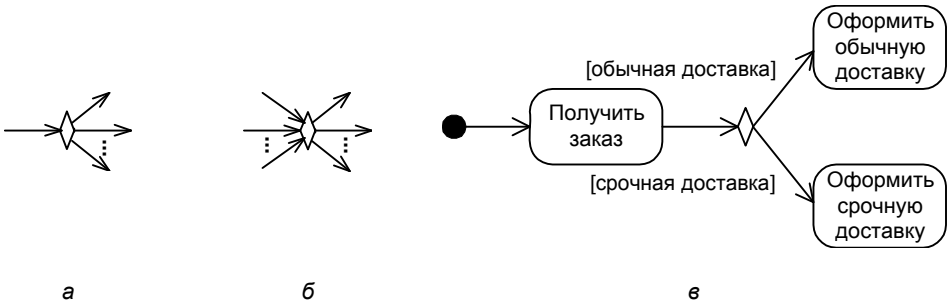


Рис. 8.8. Варианты изображения узла решения: *а* — символом ромба; *б* — объединение функциональности с узлом слияния; *в* — пример

Каждая из выходных дуг должна иметь некоторое сторожевое условие, которое при оценивании должно возвращать значение "истина" или "ложь". При оценивании всех сторожевых условий узла решения только одно из них может принимать значение "истина". Это условие будет соответствовать единственной дуге, по которой маркер проследует далее. Порядок, в котором оцениваются сторожевые условия в каждом конкретном случае, не определен.

Для некоторой выходящей из узла решения дуги может быть задано сторожевое условие с ключевым словом `else`. Это сторожевое условие является допустимым для любого маркера, если этот маркер не принят ни одной другой дугой, выходящей из этого узла решения.

Если для узла решения при оценивании оказываются справедливыми более одного сторожевого условия, то семантика такого поведения в языке UML 2.0 не определена, поскольку среди выходящих

дуг возникает состязание за прием маркера. При отсутствии дополнительной спецификации это может привести к несостоятельной (ill-formed) модели. Чтобы гарантировать выполнение только одного сторожевого условия, иногда удобно использовать процедуру проверки до первого истинного условия.

8.2.4. Узел слияния

Узел слияния (merge node) является узлом управления, который соединяет вместе несколько альтернативных потоков.

Узел слияния имеет несколько входящих дуг и одну выходящую дугу. Он не используется для синхронизации параллельных потоков, но производит один поток из нескольких, как правило, альтернативных потоков. Дуги, входящие в узел слияния и выходящие из него, все должны быть или потоками управления, или потоками объектов. Все маркеры, предлагаемые на входящих дугах, предлагаются выходящей дуге. При этом не существует никакой синхронизации потоков или соединения маркеров.

Узел слияния изображается символом ромба (рис. 8.9, *а*). Функциональность узла слияния и рассмотренного ранее узла решения может быть объединена с использованием одного символа ромба (рис. 8.8, *б*). В качестве примера можно привести фрагмент диаграммы деятельности, на которой узел слияния изображен после действий `Купить товар` и `Изготовить товар`. Когда завершится любое одно из них, то управление будет передано к действию `Доставить товар`. При этом действие `Доставить товар` будет вызвано только один раз. Если по какой-либо причине оба этих действия завершатся одновременно, то действие `Доставить товар` будет вызвано дважды (рис. 8.9, *б*).

Рис. 8.9. Варианты изображения узла слияния: *а* — символом ромба; *б* — объединение функциональностей узла решения и узла слияния

В большинстве случаев узлы слияния используются для корректного соединения нескольких альтернативных потоков в один поток. Например, если узел решения расположен после рассматриваемого далее узла разделения, то два потока, выходящие из решения, необходимо слить в один до того, как они поступят в узел соединения. В противном случае узел соединения будет ожидать оба потока, хотя только один из них его достигнет.

8.2.5. Узел разделения

Узел разделения (fork node) является узлом управления, который расщепляет поток на несколько параллельных потоков.

Узел разделения имеет одну входящую дугу и несколько выходящих дуг. Дуги, входящие в узел разделения и выходящие из него, должны быть все или потоками управления, или потоками объектов. Маркеры, достигающие узла разделения, дублируются во все выходящие дуги. Когда предлагаемый маркер признается на всех выходящих дугах, создаются дубликаты этого маркера, и один экземпляр проходит по каждой дуге.

Узел разделения изображается простым отрезком линии (рис. 8.10, *а*). Функциональность узла разделения и рассматриваемого далее узла соединения может быть объединена с использованием одного отрезка линии (рис. 8.10, *б*). Это эквивалентно изображению узла соединения со всеми изображенными входящими дугами и одной выходящей дугой, и узла разделения, который имеет одну входящую и все изображенные выходящие дуги.

В качестве примера можно привести фрагмент диаграммы деятельности, на которой изображен узел разделения, который следует за действием `Принять заказ`. Когда заканчивается действие `Принять заказ`, узел разделения отправляет управление как в действие `Заполнить заказ`, так и в действие `Послать счет`, тем самым специфицируя их параллельное выполнение (рис. 8.10, *в*).

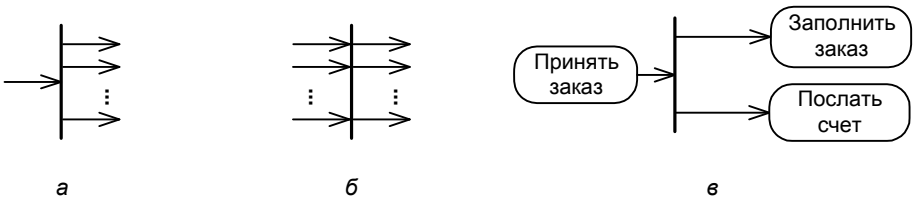


Рис. 8.10. Варианты изображения узла разделения: *а* — простым отрезком линии; *б* — объединение функциональностей узла разделения и узла соединения; *в* — пример

Если выходящие дуги не могут принять маркер по причине отказа их целей принять его, то экземпляр такого маркера помещается в неявную очередь

FIFO. Там он хранится до тех пор, пока сможет быть принят соответствующей целью. В течение этой паузы выходящие дуги не принимают маркер.

Дуги, выходящие из узла разделения, дополнительно могут иметь сторожевые условия, при невыполнении которых могут возникать паузы с передачей маркеров по этим дугам. В данном случае предполагается, что никакие из находящихся далее узлов соединения не зависят от прохода маркеров, передающихся через дугу со сторожевым условием. Если этого исключить нельзя, то необходимо ввести узел решения с последующим узлом слияния.

8.2.6. Узел соединения

Узел соединения (join node) является узлом управления, который синхронизирует несколько потоков.

Узел соединения имеет несколько входящих дуг и одну выходящую дугу. Узлы соединения могут иметь дополнительную логическую спецификацию условий, при выполнении которых они должны генерировать маркер на выходе. Если для узла соединения существуют маркеры во всех его входящих дугах, то выходящей дуге предлагаются маркеры согласно следующим правилам:

- если все маркеры, предлагаемые на входящих дугах, являются маркерами управления, то выходящей дуге предлагается один маркер управления;
- если часть маркеров, предлагаемых на входящих дугах, являются маркерами управления, а другие являются маркерами данных, то выходящей дуге предлагаются только маркеры данных. Они предлагаются выходящей дуге в том же порядке, в каком предлагаются на входе этого узла соединения.

Узел соединения изображается простым отрезком линии (рис. 8.11, *а*). Функциональность узла соединения и рассмотренного ранее узла разделения может быть объединена с использованием одного отрезка линии (рис. 8.11, *б*). В качестве примера можно привести фрагмент диаграммы деятельности, на которой изображен узел соединения для синхронизации окончания выполнения действий `Отправить заказ` и `Послать подтверждение`. Только когда оба эти действия завершатся, маркер управления будет передан действию `Закрыть заказ` (рис. 8.11, *в*).

Рядом с узлом соединения может быть изображена дополнительная спецификация (рис. 8.12, *а*). Спецификация узла соединения может содержать имена входящих дуг, чтобы ссылаться на маркер, который мог бы быть предложен по этой дуге во время оценивания логического условия. В качестве примера

спецификации узла соединения можно привести фрагмент диаграммы деятельности автомата по продаже напитков. Этот фрагмент гарантирует, что напиток будет выдан только в том случае, если он выбран, и в автомат опущено необходимое количество монет (рис. 8.12, *б*).

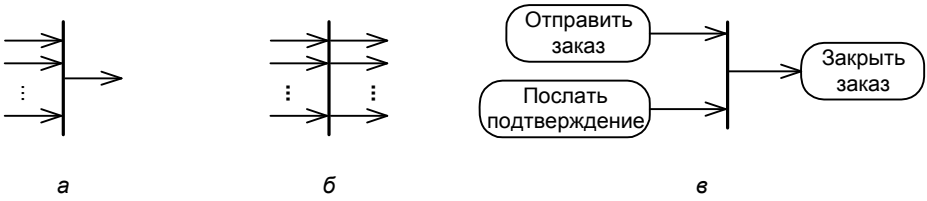


Рис. 8.11. Примеры изображения узла соединения: *а* — простым отрезком линии; *б* — объединение функциональностей узла соединения и узла разделения; *в* — пример

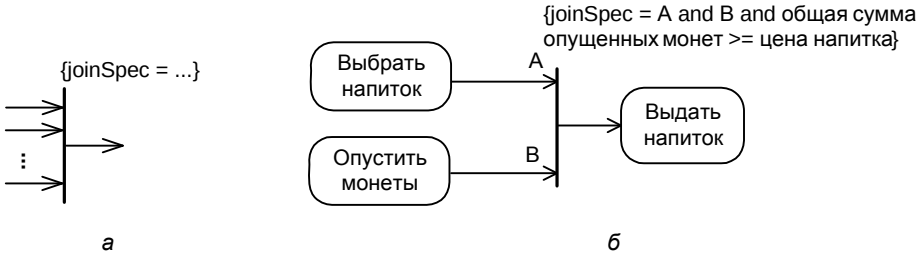


Рис. 8.12. Примеры изображения узла соединения с дополнительной спецификацией: *а* — способ представления; *б* — пример

Спецификация узла соединения оценивается всякий раз, когда по любой входящей дуге предлагается новый маркер. Начатое оценивание логического условия не прерывается никакими другими новыми маркерами, предлагаемыми в ходе оценивания. Другими словами, параллельные оценивания условий спецификации запрещены. При этом любые маркеры, предлагаемые выходящей дуге, должны быть приняты или отвергнуты до того, как любые другие маркеры будут предложены выходящей дуге. Если маркеры отвергаются для прохода, они более не предлагаются выходящей дуге.

Маркеры объектов с одной и той же индивидуальностью, предлагаемые выходящей дуге, могут быть объединены в один маркер. Другие правила могут уточнять условия следования маркеров вдоль выходящей дуги. Например, если выходящая дуга направлена в узел объекта, который достиг своей верхней границы, то маркер не может быть отправлен по этой дуге.

8.3. Специальные действия

В языке UML 2.0 определено множество специальных действий для моделирования различного поведения, такого как создание и разрушение объектов и связей, чтения и записи переменных, и выполнения целого ряда других задач. Среди этих действий наиболее часто на практике используются действия передачи и приема сигнала, которые рассматриваются далее.

8.3.1. Действие передачи сигнала

Действие передачи сигнала (send signal action) является действием, которое на основе своих входов создает экземпляр сигнала и передает его объекту цели.

Результатом передачи сигнала может быть начало выполнения некоторой деятельности или срабатывание перехода некоторого конечного автомата. При этом отправитель сигнала продолжает свое выполнение, не заботясь о результате. Любые ответные сообщения не передаются отправителю или игнорируются им.

Действие передачи сигнала обозначается с помощью выпуклого пятиугольника, в котором записывается в качестве имени тип сигнала (рис. 8.13, *а*). В качестве примера можно привести фрагмент, иллюстрирующий часть потока работ по обработке заказов, при выполнении которого передаются два сигнала. После выполнения действия `Создать заказ` на склад передается сигнал, чтобы уточнить наличие товара. После этого создается счет-фактура, по завершении этого действия клиенту передается уведомление (рис. 8.13, *б*).

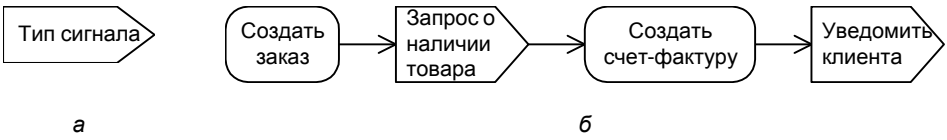


Рис. 8.13. Примеры изображения действий передачи сигнала: *а* — с помощью выпуклого пятиугольника; *б* — пример

При выполнении деятельности экземпляр сигнала передается объекту цели только в том случае, если закончено выполнение его входного действия. Объект цели может быть как локальным, так и удаленным. В ходе передачи экземпляр сигнала может быть скопирован, так что сохранение его индивидуальности не является необходимым. Способ передачи экземпляра сигнала, величина времени, требуемая на его передачу, порядок, в котором экземпляры

сигналов могут достигать различных целей, и путь, по которому достигаются эти цели, в языке UML 2.0 не определенны.

Действие передачи сигнала может оказывать непосредственное влияние на действие приема события, которое рассматривается далее.

8.3.2. Действие приема события

Действие приема события (accept event action) является действием, которое ожидает наступление некоторого события.

Действие приема события предназначено для обработки локальных или глобальных событий в ходе выполнения деятельности. Обработку действия приема события выполняет тот объект, который владеет деятельностью. Наступления событий обнаруживается объектами независимо от выполняющихся действий, при этом все наступившие события хранятся этим объектом. В общем случае порядок обнаружения событий на диаграмме деятельности не определен. В частном случае действие приема события может ожидать приема некоторого сигнала.

Действие приема события обозначается вогнутым пятиугольником (рис. 8.14, *а*). В качестве примера можно привести фрагмент, иллюстрирующий часть потока работ по обработке заказов, при выполнении которого возможна отмена заказа. Приемом события является `Запрос на отмену заказа`, который поступает при отмене клиентом заказа. Прием этого события в форме сигнала служит причиной вызова действия `Отменить заказ` (рис. 8.14, *б*).

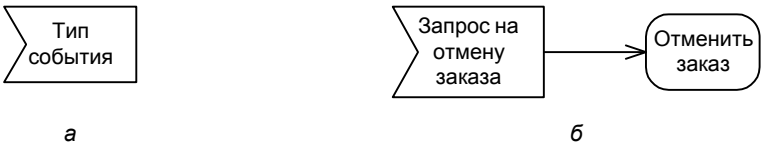


Рис. 8.14. Примеры изображения действий приема события: *а* — с помощью вогнутого пятиугольника; *б* — пример

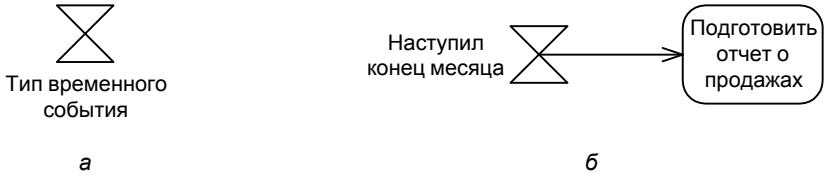
Действие приема события выполняется следующим образом. Если объект обнаружил наступление события, совпадающее с одним из действий приема события, и наступление этого события не было принято другими действиями, то действие приема данного производит на своем выходе маркер управления. Этот маркер становится доступным для всех его выходных дуг и, следовательно, может активизировать выполнение других действий. Если объект не обнаружил совпадение наступившего события ни с одним действием приема

события, то любые выходные действия будут ожидать начала своего выполнения до тех пор, пока не наступят соответствующие события.

При моделировании параллельных процессов несколько действий приема события могут состязаться за доступ к одному наступившему событию. При этом только одно действие приема может принять это наступившее событие, даже если данное наступление события будет удовлетворять нескольким параллельно выполняющимся действиям.

Специальным случаем действия приема события является действие приема или *ожидания временного события* (accept time event action), которое обозначается в форме песочных часов (рис. 8.15, *а*). Если наступившее событие является временным событием, то объект должен зафиксировать значение момента времени, когда наступило соответствующее событие. Это значение становится доступным для всех действий, которыми владеет моделируемый объект.

В качестве примера можно привести фрагмент, иллюстрирующий действие приема временного события `Наступил конец месяца`, которое генерирует маркер управления в конце каждого месяца (рис. 8.15, *б*). Поскольку в этом фрагменте для действия ожидания временного события не существует входящих дуг, то оно будет активным до тех пор, пока является активной содержащая его деятельность.



а *б*

Рис. 8.15. Примеры изображения действий приема временного события: *а* — в виде песочных часов; *б* — пример

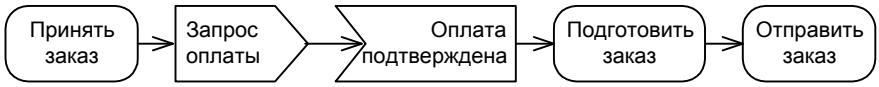


Рис. 8.16. Примеры действий передачи и приема сигналов

В качестве примера можно привести диаграмму деятельности бизнес-процесса обработки заказов, которая иллюстрирует использование рассмотренных понятий (рис. 8.16). Для удобства начальный и финальный узлы

не показаны. Сигнал `Запрос оплаты` передается после окончания действия `Принять заказ`. Затем деятельность ожидает приема сигнала о подтверждении оплаты. Прием сигнала `Оплата подтверждена` активизируется только после того, как будет передан запрос оплаты. До этого момента никакое подтверждение не признается. Когда подтверждение принимается, действие `Подготовить заказ` может начать свое выполнение. После окончания его выполнения управление передается действию `Отправить заказ`.

8.4. Узлы потока объектов

Для моделирования потока объектов на диаграммах деятельности предназначены различные узлы объектов и дуги потока объектов, которые рассматриваются далее.

8.4.1. Узел объекта

Узел объекта (object node) является узлом абстрактной деятельности, которая служит частью определяющего потока объектов в деятельности.

Узел объекта является узлом деятельности, который указывает на то, что экземпляр отдельного класса, возможно в некотором состоянии, может быть доступным в моделируемой деятельности. Узлы объектов могут быть использованы по-разному, в зависимости от того, откуда и куда текут объекты. Дополнительно узлы объектов могут осуществлять выбор маркеров, иметь ограничение на количество маркеров, а также спецификацию состояний для маркеров объектов.

Узлы объектов изображаются прямоугольниками, внутри которых размещается имя (см. рис. 8.1, *в*). Имя может содержать только имя типа узла объекта или имя и тип узла в формате "имя : тип". Узлы объектов, экземпляры которых являются множествами, изображаются аналогично. Узел объекта для маркеров объектов, находящихся в специальном состоянии, дополнительно содержит спецификацию этого состояния, которая записывается в прямых скобках ниже имени типа (рис. 8.17, *а*). Узел объекта для маркеров, содержащих множества объектов различных типов, отображает имена всех этих объектов (рис. 8.17, *б*). Узлы объектов с сигналом в качестве типа изображаются с помощью специального символа, внутри которого записывается имя типа сигнала (рис. 8.17, *в*).

Для узлов объектов дополнительно могут быть указаны некоторые ограничения, которые записываются в фигурных скобках ниже узла объекта. Так, например,

узел объекта с ограниченной верхней границей должен содержать явную спецификацию этой границы (рис. 8.18, *а*). Это значение определяет максимальное количество маркеров, которые могут быть переданы в узел. При этом объекты не могут втекать в узел, если достигнута его верхняя граница.

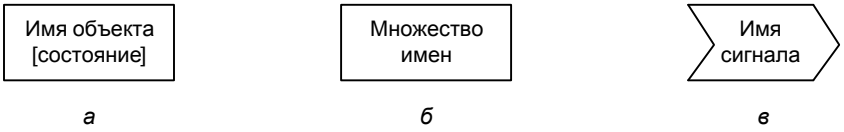


Рис. 8.17. Варианты изображения узлов объектов:
а — узел объекта для маркеров объектов;
б — узел объекта для маркеров, содержащих множества объектов различных типов;
в — узел объектов с сигналом в качестве типа

Рис. 8.18. Варианты изображения узлов объектов с дополнительными ограничениями:
а — узел объекта с ограниченной верхней границей;
б — с указанием порядка управления маркерами объектов;
в — с ключевым словом `<<selection>>`

Узел объекта может специфицировать порядок управления маркерами объектов, отличающимся от FIFO, в форме дополнительной спецификации (рис. 8.18, *б*). Порядок указывает, являются ли упорядоченными и как упорядочены маркеры в узле объекта для выбора, чтобы двигаться по дугам, выходящим из узла объекта. Вид упорядочения узла объекта, который определяет порядок очереди в узле, может принимать одно из следующих значений: `unordered` (неупорядоченный), `ordered` (упорядоченный), `LIFO` (Last In, First Out — последним пришел, первым вышел), FIFO (First In, First Out — первым пришел, первым вышел). По умолчанию принимается значение FIFO.

Дополнительная спецификация выбора маркеров помечается ключевым словом `<<selection>>`, размещенным в символе примечания и присоединенным к символу узла (рис. 8.18, *в*). Если узел объекта имеет спецификацию выбора, то узел объекта имеет порядок маркеров, и наоборот.

Все дуги, входящие в узлы объектов или выходящие из них, должны быть дугами потока объектов. Узлы объектов могут содержать только маркеры объектов, которые соответствуют типу этого узла. Если тип узла не специфицирован, то маркеры объектов могут быть любого типа. В узле объекта в одно и то же время может находиться несколько маркеров, содержащих одно и то же значение. Это позволяет моделировать передачу и преобразование данных. Маркер из узла объекта может проследовать только по одной из выходящих дуг.

Узел объекта не может содержать больше маркеров, чем его верхняя граница. Верхняя граница узла объекта по умолчанию равна верхней границе ближайших расположенных до или после узлов объектов, которые не имеют промежуточных узлов действий. Верхняя граница может иметь в качестве конкретного значения некоторое натуральное число. Верхняя граница с символом * означает, что она является неограниченной.

Порядок узла объекта специфицирует последовательность, в которой маркеры в узле предлагаются выходящим дугам. Это может быть задано для того, чтобы маркеры не налагались друг на друга, когда они проходят через узел (FIFO), или чтобы они что-либо делали (LIFO). Разработчик также может определить собственный порядок с помощью значения `ordered` и спецификации выбора, которая определяет, какой маркер из очереди должен быть предоставлен выходящим дугам.

Спецификация выбора описывает поведение, которое захватывает все маркеры на входе узла объекта и выбирает только один из них. Оно выполняется всякий раз, когда выходной дуге должен быть предложен маркер. Поскольку это поведение используется в то время, когда маркеры предлагаются выходящим дугам, оно может выполняться несколько раз для одного и того же маркера, пока этот маркер не будет принят выходными дугами.

Для узла объекта дополнительно может быть указано, что его тип соответствует маркеру управления. Это также справедливо, если для узла не специфицирован никакой тип. В этом случае с узлом объекта, имеющим тип управления, могут быть использованы дуги управления. Эта ситуация встречается на практике достаточно часто, о чем следует помнить разработчикам.

8.4.2. Центральный буфер и хранилище данных

Центральный буфер (central buffer) является узлом объекта для управления потоками из нескольких источников и мест назначения.

Центральный буфер принимает маркеры из расположенных до него узлов объектов и отправляет их дальше в расположенные после него узлы объектов. Центральный буфер не имеет непосредственных соединений с узлами действия. Хотя все узлы объектов имеют функциональность буфера, но центральные буферы отличаются тем, что они не привязаны к действиям, как рассматриваемые далее контакты и параметры.

Центральный буфер изображается символом обычного узла объекта с дополнительным ключевым словом `<<centralBuffer>>`, записанным в форме стереотипа. В качестве примера использования центрального буфера можно рассмотреть фрагмент, который иллюстрирует изготовление деталей на двух заводах (рис. 8.19). После их изготовления они поступают в центральный буфер, а затем детали могут быть либо использованы в собственном производстве, либо проданы другим производителям. Все детали, которые не использованы, будут проданы, и наоборот, поскольку каждый маркер может проследовать из центрального узла только по одной выходящей дуге.

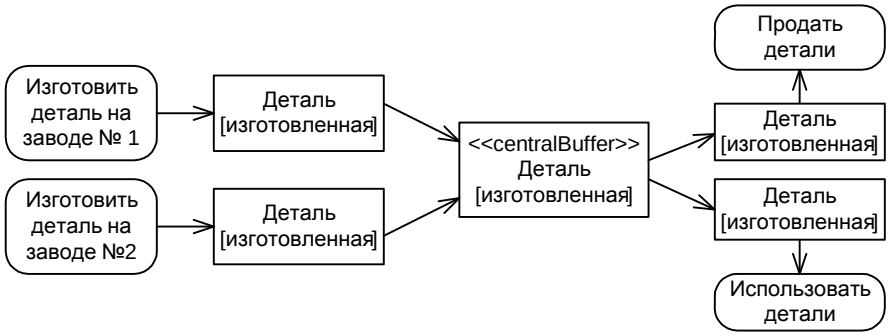


Рис. 8.19. Пример использования центрального буфера

В рассмотренном примере выбор объектов на выходе центрального буфера является недетерминированным, что хотя и допустимо, но нежелательно. В этом случае следует дополнить центральный буфер спецификацией с целью поддержки очереди объектов и исключения конкуренции за потоки объектов на своем выходе.

Для моделирования узлов данных, в которых данные хранятся постоянно и могут использоваться по мере необходимости, предназначены хранилища данных.

Хранилище данных (data store) является разновидностью центрального буфера для постоянного хранения объектов или другой информации.

Хранилище данных принимает все маркеры, которые в него входят, сохраняет их и копирует, когда они выбираются для дальнейшего использования. Если входящий маркер представляет объект, который уже имеется в хранилище, то он заменяет любой маркер в узле объекта, содержащий этот объект.

Хранилище данных изображается символом обычного узла объекта с использованием ключевого слова `<<datastore>>`, записанным в форме стереотипа. В качестве примера использования хранилища данных можно рассмотреть фрагмент, который иллюстрирует работу персонала в некоторой компании (рис. 8.20). При приеме сотрудника на работу информация о нем заносится в базу данных, которая моделируется в форме хранилища данных. При этом сразу выполняется его назначение на некоторую должность. В дополнение к этому каждое полугодие происходит пересмотр назначения всех сотрудников. Во всех этих случаях информация о сотрудниках из базы не удаляется.

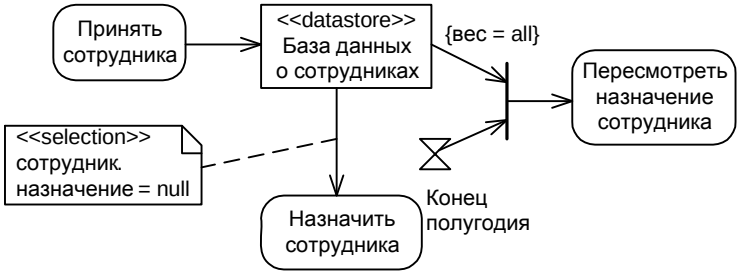


Рис. 8.20. Пример использования хранилища данных

Копирование маркеров в хранилище данных выполняется всякий раз, когда они выбираются для следования по выходным дугам. Тем самым ситуация представляется таким образом, что маркеры как бы никогда не покидают хранилище данных. Хотя это и исключает конкуренцию выходных дуг за маркеры, но не позволяет моделировать другие ресурсы в бизнес-процессах.

8.4.3. Входные и выходные контакты объектов

Входной контакт (input pin) является узлом объекта, который принимает значения от других действий в форме потока объектов. *Выходной контакт (output pin)* является узлом объекта, который поставляет значения другим действиям в форме потока объектов.

На диаграммах деятельности контакты изображаются в форме маленьких прямоугольников, которые присоединяются к символам действий. Как правило,

входные контакты изображаются слева от действия (рис. 8.21, *а*), а выходные контакты — справа (рис. 8.21, *б*). Чтобы дополнительно визуализировать вид контакта, внутри символа контакта может быть помещена необязательная стрелка. В этом случае входные контакты имеют стрелку, направленную в действие (рис. 8.21, *в*), а выходные контакты имеют стрелку, направленную от действия (рис. 8.21, *г*).

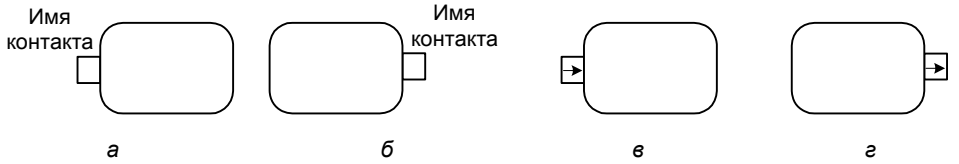


Рис. 8.21. Варианты изображения входных и выходных контактов:
а, *в* — входной контакт слева от действия;
б, *г* — выходной контакт справа от действия

Рядом с контактом может быть указано имя контакта. В качестве имени часто указывают тип объекта или тип данных, которые протекают через этот контакт. В общем случае имя контакта может быть записано в форме "имя : тип".

В языке UML 2.0 передача объекта от одного действия к другому может быть изображена несколькими способами. Во-первых, символ объекта можно соединить дугами с соответствующими действиями (см. рис. 8.5). Во-вторых, выходной контакт одного действия можно соединить с входным контактом другого действия, при этом имена контактов должны совпадать с именем объекта (рис. 8.22, *б*). Таким образом, фрагмент диаграммы, изображенный на рис. 8.5, *а*, может быть представлен так, как показано на рис. 8.22, *б*. Эти два фрагмента являются семантически эквивалентными.

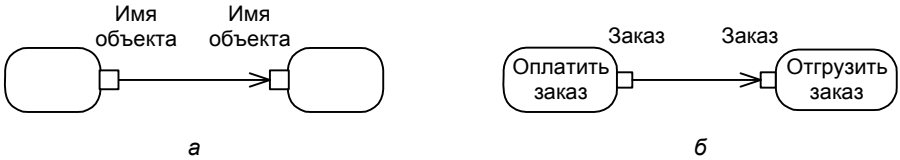


Рис. 8.22. Вариант изображения передачи объекта между действиями с контактами:
а — способ представления; *б* — пример

Для представления потока объектов между контактами рядом с символом контакта размещается специальная строка текста: `{stream}` (рис. 8.23, *а*). Если она пропущена, то по умолчанию принимается `{nonstream}`, т. е. передача индивидуального объекта.

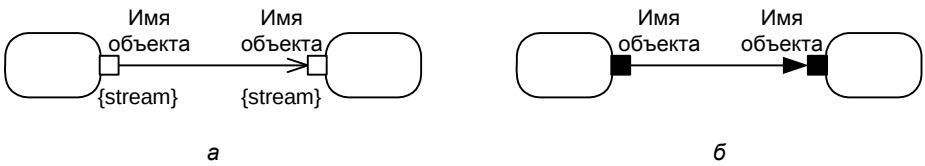


Рис. 8.23. Варианты изображения передачи потока объектов между действиями с контактами: *а* — с дополнительной строкой текста `{stream}`; *б* — без строки текста

Альтернативно поток объектов между контактами может быть изображен без специальной строки текста. В этом случае контакты изображаются в форме заполненных прямоугольников, а узлы действий соединяются непрерывными линиями с треугольной стрелкой (рис. 8.23, *б*).

В общем случае на диаграммах деятельности могут также использоваться контакты управления. Они изображаются аналогично контактам объектов с дополнительной строкой текста `{control}`, размещенной рядом с символом контакта. При этом допускается не показывать контакты на диаграмме вовсе, даже если они и существуют в модели.

8.4.4. Узел параметра деятельности

Для интеграции деятельности с внешними потоками объектов или другими элементами модели предназначены узлы параметра деятельности.

Узел параметра деятельности (activity parameter node) является узлом объекта для моделирования входов и выходов деятельности.

Параметры деятельности являются узлами объектов и предназначены для моделирования входов в деятельность и выходов из нее. Тип узла параметра деятельности должен соответствовать типу параметра деятельности. Узлы входных параметров деятельности не имеют входящих дуг и имеют одну или более выходящих дуг. Узлы выходных параметров деятельности не имеют выходящих дуг и имеют одну или более входящих дуг.

Узлы параметров деятельности изображаются в форме символа прямоугольника на границе символа деятельности (см. рис. 8.6, *а*). Внутри символа узла параметра деятельности может быть указано его имя, а рядом с ним — дополнительная спецификация соответствующего параметра. При вызове деятельности на ее входах помещаются значения, представляющие собой маркеры

на узлах входных параметров деятельности. После завершения деятельности на ее выходах помещаются значения, представляющие собой маркеры на узлах выходных параметров деятельности.

Вызов и выполнение некоторой деятельности с параметрами выполняется в соответствии со следующими правилами:

- для вызываемой деятельности должны быть достигнуты все необходимые непотоковые входы;
- если все входы в деятельность являются входами потока объектов, то для начала выполнения деятельности должен быть достигнут, по крайней мере, один из узлов входных параметров этой деятельности;
- во время выполнения деятельности на ее узлах входных параметров могут поглощаться дополнительные маркеры, а на ее узлах выходных параметров — отправляться;
- для завершения деятельности должны быть достигнуты все специфицированные узлы выходных параметров этой деятельности.

Таким образом, для завершения деятельности необходимо, чтобы во всех узлах выходных параметров этой деятельности находились маркеры. Однако если некоторые узлы выходных параметров имеют спецификацию пустого маркера `null`, то для завершения деятельности они могут оставаться пустыми. Приведенные правила позволяют выполнять вызовы одной деятельности из другой деятельности.

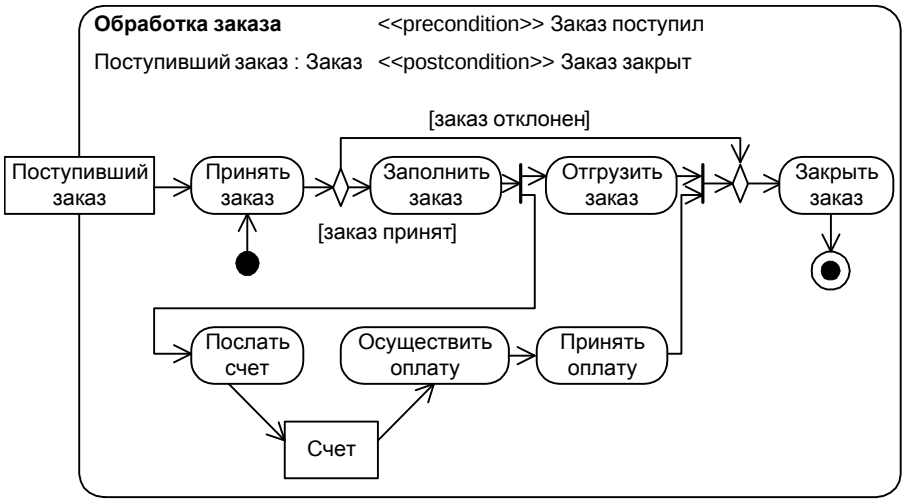


Рис. 8.24. Пример деятельности с входным параметром

В качестве примера можно рассмотреть диаграмму деятельности `Обработка заказа`, которая иллюстрирует использование узла входного параметра для этой деятельности (рис. 8.24). Для вызова этой деятельности необходимо наличие маркера в узле входного параметра, что соответствует наличию поступившего заказа. После начала деятельности этот маркер изымается из этого узла и передается в выполнение действия `Принять заказ`.

После выполнения одной из двух альтернативных ветвей после передачи управления к узлу финала данная деятельность может быть завершена. Поскольку никакие выходы для рассматриваемой деятельности не специфицированы, то никакие маркеры объектов в них размещать не требуется. Данная деятельность также имеет предусловие и постусловие, которые формально специфицируют ее начало и окончание. Дополнительно на диаграмме указано, что входной параметр `Поступивший заказ` имеет тип `Заказ` и может рассматриваться как атрибут соответствующего класса.

Рассмотренные правила вызова и выполнения деятельности могут быть уточнены разработчиками. Так, например, можно дополнительно специфицировать узлы входных параметров, которые могут не иметь маркеров на момент начала деятельности. С этой целью следует явно указать нижнюю и верхнюю границы значений для количества маркеров, передаваемых параметру во время выполнения. Тогда нижняя граница 0 будет означать, что данный параметр является необязательным. В этом случае деятельность, использующая необязательный параметр, может начать свое выполнение без наличия для него значений. Нижняя граница больше 0 означает, что для деятельности требуются значения данного параметра.

8.4.5. Множество параметров

Для спецификации различных потоков маркеров при вызове деятельности используется множество параметров.

Множество параметров (parameter set) предназначено для моделирования отдельного альтернативного потока объектов на входе или выходе деятельности.

Обычно вызов деятельности требует задания значений для всех ее входных параметров, а после выполнения деятельности все ее выходные параметры должны получить некоторые значения. Другими словами, несколько потоков объектов, входящих или выходящих после вызова деятельности, обрабатываются обычно как условия "И". Однако в отдельных случаях наличие значений для всех входных или выходных параметров деятельности не является

необходимым. Для спецификации подобных ситуаций используется множество параметров.

Если для некоторой деятельности специфицировано только одно множество входных параметров, то для вызова этой деятельности необходимо наличие маркеров объектов для каждого из входных параметров из этого множества. Если же для деятельности специфицировано несколько множеств входных параметров, то для вызова деятельности необходимо наличие маркеров объектов для входных параметров только для одного такого множества.

Множество параметров деятельности изображается размещением узлов входных или выходных параметров внутри прямоугольника. В этом случае отдельному множеству параметров будет соответствовать отдельный прямоугольник. В качестве общего примера рассмотрим деятельность с двумя множествами входных параметров и с двумя множествами выходных параметров (рис. 8.25, *а*). Для отдельного выполнения этой деятельности необходимо наличие двух маркеров для верхней группы узлов входных параметров или трех маркеров для нижней группы узлов входных параметров. Таким образом, данная деятельность может начаться, только когда одна или другая группа имеет полное множество входных потоков маркеров. После выполнения этой деятельности маркеры будут помещены либо в верхнюю группу, либо в нижнюю группу узлов выходных параметров.

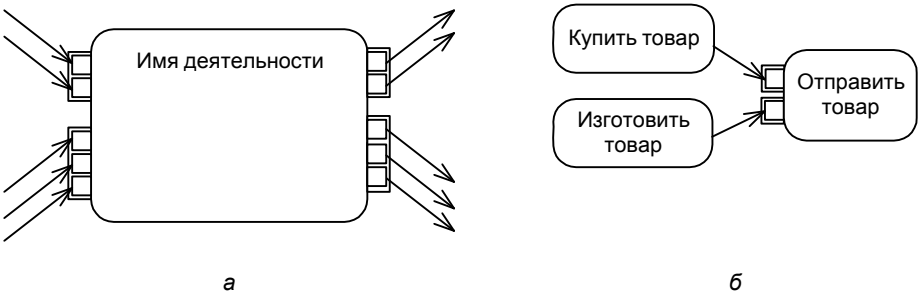


Рис. 8.25. Варианты изображения множества параметров: *а* — с двумя множествами входных параметров и двумя множествами выходных параметров; *б* — пример

Другим примером может служить фрагмент диаграммы, изображенный на рис. 8.25, *б*. Здесь деятельность `Отправить товар` начинается всякий раз, когда она принимает купленный товар или изготовленный товар.

Таким образом, однократное выполнение деятельности с несколькими множествами входных параметров может принимать входные параметры только из одного множества параметров. Соответственно, однократное выполнение

деятельности с несколькими множествами выходных параметров может отправлять выходные параметры только в одно множество параметров. Тем самым отдельное множество параметров ведет себя как полное множество входов или выходов для деятельности. При этом другие множества параметров для этой деятельности не являются необходимыми для ее вызова.

8.5. Специальные регионы

При моделировании бизнес-процессов в нотации UML 1.*x* наибольшее распространение получил вариант представления диаграмм деятельности с так называемыми *дорожками* (swimlanes). Хотя в языке UML 2.0 нотация дорожек сохранена, но они стали изображаться горизонтально и называться более общим понятием — разбиением деятельности.

8.5.1. Разбиение деятельности

Разбиение деятельности (activity partition) — элемент модели, предназначенный для группировки действий, которые относятся к одной деятельности и имеют некоторую общую характеристику.

На практике при моделировании бизнес-процессов разбиения наиболее часто соответствуют организационным единицам. Разбиения также используются для графического изображения ресурсов совместно с узлами деятельности. В языке UML 2.0 разбиения могут быть иерархическими и многомерными.

Отдельное разбиение деятельности изображается в форме двух горизонтальных или вертикальных параллельных линий с именем разбиения, помещенным в прямоугольник на одном из его концов (рис. 8.26, *а*). Любые узлы и дуги, помещенные между этими линиями, рассматриваются как принадлежащие этому разбиению. Это изображение по-прежнему называют разбиением с использованием нотации дорожки.

Иерархические разбиения представляются в форме дополнительного разбиения некоторого исходного разбиения или родителя (рис. 8.26, *б*). Такое изображение называют также разбиением с использованием нотации иерархических дорожек.

Диаграммы деятельности могут также содержать разбиения в нескольких измерениях (рис. 8.26, *в*), в которых каждая ячейка дорожки является пересечением нескольких разбиений. Такое разбиение соответствует нотации многомерных иерархических дорожек. Например, деятельность может иметь одно измерение для указания местоположения, а другое — для указания организационных подразделений, в которых выполняются ее отдельные действия.

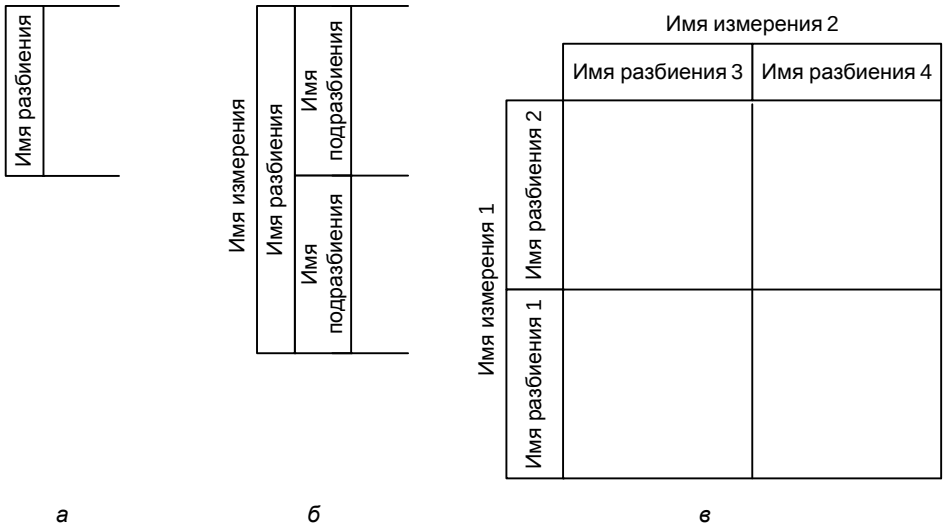


Рис. 8.26. Варианты изображения разбиений деятельности:
а — разбиением с использованием нотации дорожки;
б — разбиение с использованием нотации иерархических дорожек;
в — разбиение с использованием нотации многомерных иерархических дорожек

При использовании разбиений на диаграмме деятельности следует руководствоваться следующими правилами:

- любой узел или дуга деятельности не может одновременно принадлежать двум или более разбиениям в одном и том же измерении;
- разбиения не влияют на потоки маркеров, за которые несут ответственность экземпляры классификаторов, представленными отдельными разбиениями;
- разбиения одного измерения и уровня вложенности должны быть представлены частями внутренней структуры одного и того же классификатора;
- разбиение может быть представлено атрибутом, а его подразбиения — значениями этого атрибута. Например, разбиение может представлять местоположение, в котором выполняется деятельность, а подразбиения могут указывать на специальные значения для такого атрибута, как название конкретного местоположения или адрес;
- если разбиение имеет некоторое измерение, то оно не может содержаться ни в каком другом разбиении.

В качестве примера рассмотрим диаграмму деятельности по обработке заказов с тремя разбиениями (рис. 8.27). Верхнее разбиение содержит подмножество

действий, за которые отвечает отдел заказов; среднее разбиение — бухгалтерия, а нижнее — действия, выполнение которых ожидается от клиента. Поскольку клиент является внешней сущностью, то это явно отмечено при помощи ключевого слова `<<external>>`.

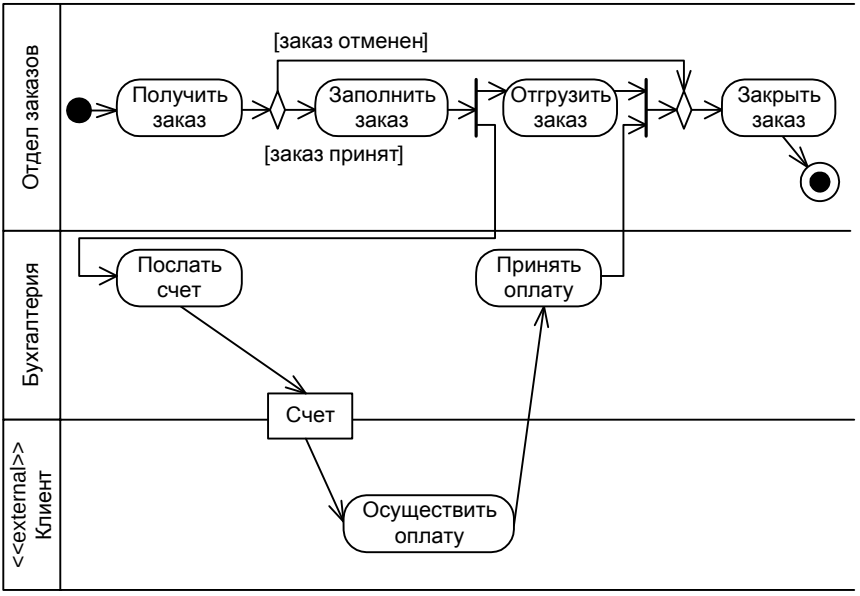


Рис. 8.27. Пример разбиений деятельности с использованием дорожек

Следующий пример иллюстрирует использование многомерных дорожек (рис. 8.28). Действия `Получить заказ` и `Заполнить заказ` выполняются экземпляром класса `Менеджер по продажам`, расположенным в `Москве`. Действия `Отгрузить заказ` и `Закрыть заказ` выполняются экземпляром класса `Менеджер по продажам`, расположенным в `Санкт-Петербурге`. Аналогично, действие `Послать счет` выполняет бухгалтер из Москвы, а действие `Принять оплату` — бухгалтер из Санкт-Петербурга.

Хотя действие `Осуществить оплату` содержится в ячейке дорожки `Москва/Бухгалтер`, ее исполнитель и местоположение в этом разбиении не специфицированы. Стереотип `<<external>>` указывает лишь на внешний характер этого действия.

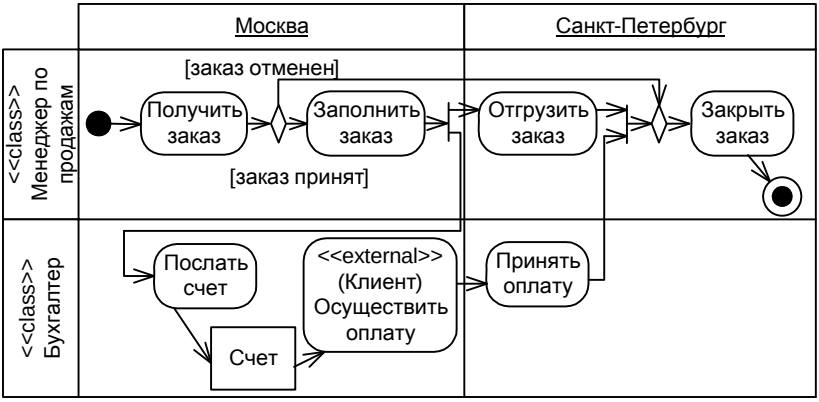


Рис. 8.28. Пример разбиений деятельности с использованием многомерных дорожек

8.5.2. Регион прерываемой деятельности

Для моделирования прекращения всех потоков в деятельности предназначены регионы прерываемой деятельности или прерываемые регионы.

Регион прерываемой деятельности (interruptible activity region) является элементом модели, который поддерживает прекращение всех потоков маркеров в некоторой деятельности.

Регион прерываемой деятельности содержит узлы и дуги, при этом одна или несколько дуг являются дугами прерываемой деятельности и выходят из этого региона. Если маркер покидает прерываемый регион по некоторой дуге прерываемой деятельности, то все маркеры и действия в этом регионе прекращаются. Дуга прерываемой деятельности называется также *прерывающей дугой* (interrupting edge). При этом узел, являющийся источником для прерывающей дуги региона, должен находиться в самом этом регионе. Узел, являющийся целью для прерывающей дуги региона, должен находиться за пределами региона, но в той же деятельности, которая содержит этот регион.

Регион прерываемой деятельности обозначается пунктирным прямоугольником с закругленными углами, внутри которого изображаются узлы, содержащиеся в этом регионе. Прерывающая дуга изображается как дуга деятельности со стрелкой в форме молнии (рис. 8.29, *а*). Допускается также изображать прерывающую дугу в форме зигзага рядом с прямой линией для дуги деятельности (рис. 8.29, *б*).

Рис. 8.29. Варианты изображения региона прерываемой деятельности и прерывающей дуги: *а* — прерывающая дуга в форме стрелки "молния"; *б* — прерывающая дуга в форме зигзага рядом с прямой линией для дуги деятельности

Регион прерываемой деятельности прерывается, когда любой маркер проходит по прерывающей дуге. Передача маркера при этом является атомарной, т. е. переход маркера является либо полным, либо не происходит вовсе. На практике регион прерываемой деятельности используется для моделирования аварийного завершения всех потоков в некоторой деятельности.

В качестве примера рассмотрим фрагмент диаграммы деятельности по обработке заказов (рис. 8.30). Изображенный регион прерываемой деятельности содержит действие приема сигнала `Запрос на отмену заказа`, при поступлении которого вызывается действие `Отменить заказ`. Однако регион прерываемой деятельности будет покинут только во время выполнения действий по получению, заполнению или отгрузке заказа.

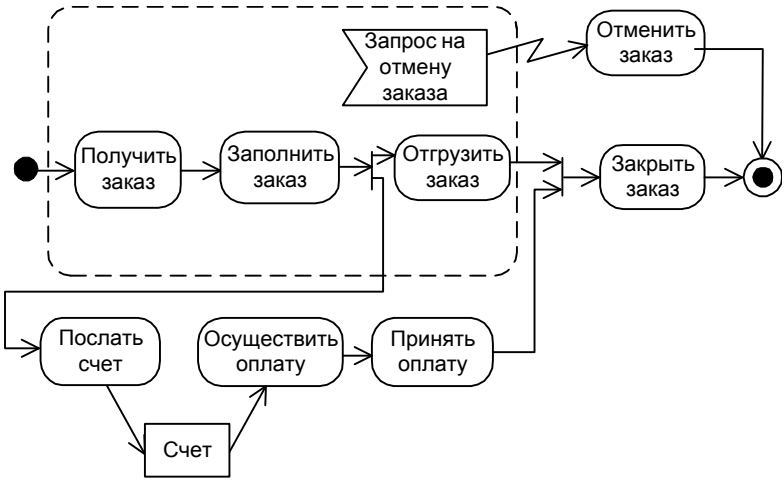


Рис. 8.30. Пример региона прерываемой деятельности

В рассмотренном примере регион прерываемой деятельности может быть расширен на действия, связанные с оплатой счета, хотя это и не изменит логику управления потоком на диаграмме. С другой стороны, действие `Отгрузить заказ` можно было бы вынести за пределы региона. Во всех этих случаях отдельное решение будет отражать модель конкретного бизнес-процесса.

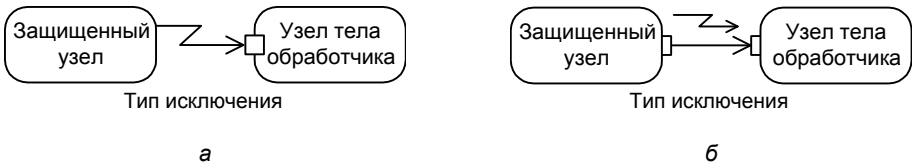
8.5.3. Обработчик исключения

Обработчик исключения (exception handler) представляет собой спецификацию деятельности, выполнение которой происходит в случае исключения в ходе выполнения некоторого защищенного узла.

С обработчиком исключения связаны два узла и одна дуга деятельности. Узел, который реагирует на некоторое исключение, называется *защищенным узлом* (protected node). Узел, который выполняется, если обработчик удовлетворяет перехваченному исключению, называется *телом обработчика* (handler body). Дуга деятельности, которая связывает защищенный узел с телом обработчика, в этом случае помечается некоторым типом исключения.

Если при выполнении деятельности в защищенном узле происходит исключение, которое совпадает с типом исключения обработчика, то обработчик перехватывает это исключение и выполняет тело обработчика.

Обработчик исключения для защищенного узла изображается в форме стрелки "молния" от границы защищенного узла к небольшому квадрату на границе тела обработчика исключения (рис. 8.31, *а*). Имя типа исключения размещается ниже этой стрелки. Небольшой квадрат является входным узлом исключения, и он находится в собственности тела обработчика. Его тип является типом данного исключения. Обработчик исключения может быть изображен с использованием прерывающей дуги форме зигзага рядом с линией дуги (рис. 8.31, *б*).



а *б*

Рис. 8.31. Варианты представления обработчика исключения:
а — обработчик исключения для защищенного узла в форме стрелки "молния";
б — обработчик исключения с прерывающей дугой в форме зигзага рядом с линией дуги

При использовании обработчиков исключений на диаграмме деятельности следует руководствоваться следующими правилами:

- тело обработчика должно иметь один вход, и этот вход является тем же, что и вход исключения;
- контакты результата тела обработчика исключения должны соответствовать по числу и типу контактам результата защищенного узла;
- защищенный узел и узел тела обработчика должны быть одного и того же уровня вложенности. В противном случае нотация может быть интерпретирована ошибочно как прерывающая дуга, которая пересекает границу;
- к одному защищенному узлу могут быть присоединены несколько обработчиков исключений, каждый со своими собственными стрелками-молниями.

При наступлении исключения все маркеры в защищенном узле блокируются, и проверяется множество обработчиков исключения. Если существует обработчик, который совпадает с типом наступившего исключения, то этот обработчик "перехватывает" данное исключение, и начинается выполнение тела обработчика. Если исключение не перехватывается никаким обработчиком, то это исключение передается далее в те узлы или деятельности, которые содержат рассматриваемый защищенный узел. Если исключение было передано до самого верхнего уровня системы, и оно не было перехвачено никаким обработчиком, то такое поведение системы является неопределенным и требует дополнительной спецификации.

Тело обработчика не имеет явно входящих или выходящих дуг и имеет такой же доступ к окружающему контексту, как и защищенный узел. Результирующие маркеры тела обработчика становятся результирующими маркерами соответствующего защищенного узла. Любые дуги управления, выходящие из защищенного узла, принимают маркеры управления по завершении выполнения тела обработчика от обработчика, перехватившего данное исключение. Когда тело обработчика завершает выполнение, оно становится таким, как если бы защищенный узел закончил свое выполнение.

В качестве примера использования обработчика исключения рассмотрим диаграмму деятельности по обработке заказов на продажу товара (рис. 8.32). При выполнении действия `Оформить заказ` может оказаться, что поступившая заявка содержит неполные данные. В этом случае генерируется исключение о том, что заявка неполная, и используется соответствующий обработчик исключения. После выполнения тела обработчика по дооформлению заказа управление передается на узел разделения. Однако если при заполнении заказа данные заявки полные, то рассматриваемое исключение не происходит, и управление сразу передается на узел разделения.

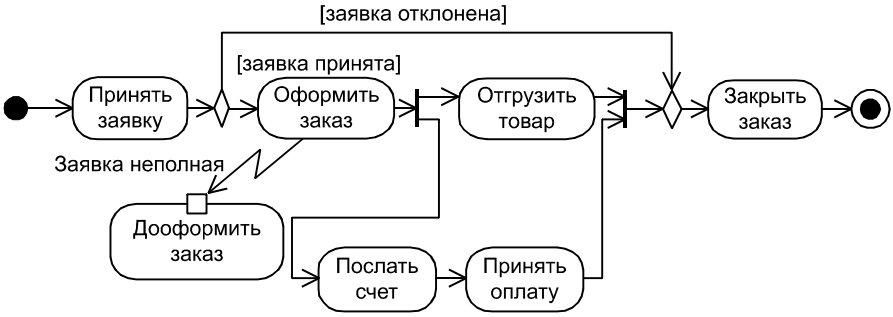


Рис. 8.32. Пример использования обработчика исключения

* * *

В заключение этой главы приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм деятельности (табл. 8.1).

Таблица 8.1. *Графические узлы и пути на диаграммах деятельности*

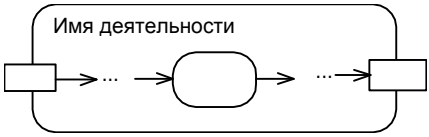
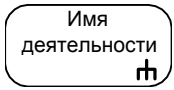
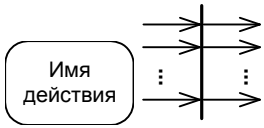
Тип графического элемента	**Нотация**	**Ссылка на подраздел**
Деятельность (activity)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Узел вызова деятельности (call activity node)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Узел действия (action node)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Начальный узел (initial node)		Начальный узел *(см. разд. 8.2.1)*
Узел финала деятельности (activity final node)		Узел финала деятельности и потока *(см. разд. 8.2.2)*
Узел финала потока (flow final node)		Узел финала деятельности и потока *(см. разд. 8.2.2)*

Таблица 8.1 *(продолжение)*

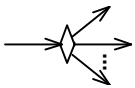
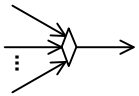
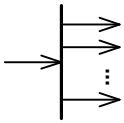
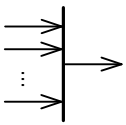
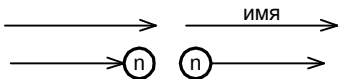
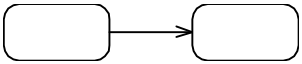
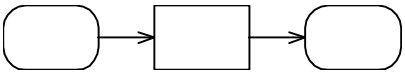
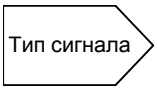
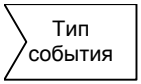
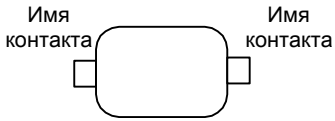
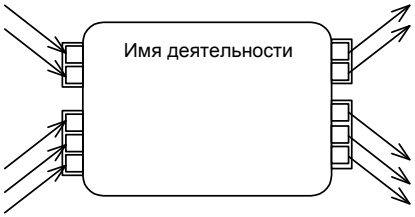
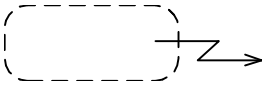
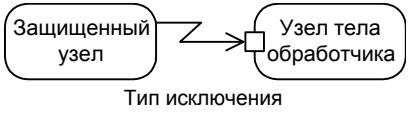
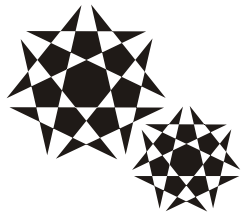
Тип графического элемента	**Нотация**	**Ссылка на подраздел**
Узел решения (decision node)		Узел решения *(см. разд. 8.2.3)*
Узел слияния (merge node)		Узел слияния *(см. разд. 8.2.4)*
Узел разделения (fork node)		Узел разделения *(см. разд. 8.2.5)*
Узел соединения (join node)		Узел соединения *(см. разд. 8.2.6)*
Узел объекта (object node)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Дуга деятельности (activity edge)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Поток управления (control flow)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Поток объектов (object flow)		Узлы и дуги деятельности *(см. разд. 8.1.2)*
Действие передачи сигнала (send signal action)		Действие передачи сигнала *(см. разд. 8.3.1)*
Действие приема события (accept event action)		Действие приема события *(см. разд. 8.3.2)*
Действие приема временного события (accept time event action)		Действие приема события *(см. разд. 8.3.2)*

Таблица 8.1 *(окончание)*

Тип графического элемента	Нотация	Ссылка на подраздел
Центральный буфер (central buffer)	<<centralBuffer>> Имя объекта [состояние]	Центральный буфер и хранилище данных *(см. разд. 8.4.2)*
Хранилище данных (data store)	<<datastore>> Имя объекта [состояние]	Центральный буфер и хранилище данных *(см. разд. 8.4.2)*
Входной контакт (input pin) и выходной контакт (output pin)		Входные и выходные контакты объектов *(см. разд. 8.4.3)*
Множество параметров (parameter set)		Множество параметров *(см. разд. 8.4.5)*
Разбиение деятельности (activity partition)		Разбиение деятельности *(см. разд. 8.5.1)*
Регион прерываемой деятельности (interruptible activity region)		Регион прерываемой деятельности *(см. разд. 8.5.2)*
Обработчик исключения (exception handler)		Обработчик исключений *(см. разд. 8.5.3)*

Глава 9



Вспомогательные диаграммы взаимодействия

Для моделирования взаимодействия в языке UML 2.0 предназначено несколько канонических диаграмм. Наиболее общей моделью взаимодействия является диаграмма последовательности, которая концентрирует внимание на взаимном обмене сообщениями между несколькими линиями жизни в контексте времени. Наличие специальных конструкций для визуализации альтернативности и параллельности, вложенности и временных характеристик делает этот тип диаграмм основным средством для моделирования взаимодействия.

Кроме диаграмм последовательности для моделирования взаимодействия могут использоваться диаграммы коммуникации, диаграммы обзора взаимодействия и временные диаграммы. Диаграммы коммуникации изображают взаимодействия на фоне некоторого архитектурного представления, при этом структурные связи между взаимодействующими линиями жизни дополняются передаваемыми по ним сообщениями. Диаграммы обзора взаимодействия являются некоторым макропредставлением диаграмм деятельности, которые позволяют представить общий поток управления при реализации взаимодействия. Временная диаграмма изображает особенности взаимодействия одной или нескольких линий жизни во времени.

Эти типы канонических диаграмм содержат ограниченное число конструкций, поэтому их вполне можно назвать вспомогательными диаграммами взаимодействия. Описание этих конструкций и особенности их использования служат предметом данной главы.

9.1. Диаграмма коммуникации (communication diagram)

Диаграмма коммуникации (communication diagram) — диаграмма, которая предназначена для представления взаимодействия в контексте внутренней архитектуры системы и передаваемых сообщений.

В языке UML 2.0 диаграмма коммуникации служит для визуализации взаимодействия между частями композитной структуры или ролями взаимодействия. В отличие от диаграммы последовательности, на диаграмме коммуникации могут быть изображены структурные отношения между элементами модели. С другой стороны, время на диаграмме коммуникации в качестве неявного измерения не изображается, поэтому для определения порядка следования сообщений необходимо указывать их порядковый номер.

Диаграмма коммуникации имеет вид графа, вершинами которого являются части композитного класса или роли взаимодействия, представленные в виде прямоугольников. Эти вершины соответствуют линиям жизни и изображаются в своем структурном контексте. Ребрами графа являются связи, по которым проходят маршруты коммуникации. Линии жизни могут обмениваться сообщениями, которые изображаются в виде небольших стрелок с некоторым именем, расположенным возле линий связей. Обычно взаимодействие, моделируемое диаграммой коммуникации, соответствует реализации некоторой операции класса или варианта использования.

Диаграмма коммуникации описывает связи и сообщения, характерные только для одного определенного контекста. Поэтому на диаграммах коммуникации могут использоваться несколько экземпляров одного и того же класса, играющих различные роли во взаимодействии. Как правило, полное описание структуры сложной системы может быть получено посредством комбинации диаграмм классов и диаграмм коммуникации. При этом характерной особенностью диаграммы коммуникации является возможность графического представления не только необходимых структурных отношений между ролями взаимодействия, но и визуализации последовательности сообщений, которые занимают центральное место в этой модели взаимодействия.

Хотя диаграммы коммуникации используются для спецификации поведения системы, но время в явном виде в них отсутствует. Поведение системы описывается посредством обмена сообщениями между линиями жизни, обработка которых позволяет достичь нужной цели или реализовать необходимую функциональность. Для аналитика или архитектора системы важным является также представление в проекте структурных связей между отдельными

экземплярами классов. Такое объединенное представление структуры и взаимодействия обеспечивает диаграмма коммуникации.

Таким образом, диаграмма коммуникации позволяет визуализировать линии жизни, которые соответствуют, как правило, экземплярам классов, связи, которые соответствуют экземплярам ассоциаций, и сообщения. При этом на диаграмме коммуникации показываются только те линии жизни, которые имеют непосредственное отношение к реализации моделируемого взаимодействия.

Следует заметить, что хотя диаграммы последовательности и диаграммы коммуникации могут представлять аналогичную информацию, но визуализируют ее различными способами. Как правило, диаграммы коммуникации в языке UML 2.0 соответствуют простым диаграммам последовательности, в которых не используются специальные механизмы структуризации, такие как фреймы взаимодействия и комбинированные фрагменты. Напомним, что в прежних версиях языка UML 1.*x* диаграммы коммуникации назывались диаграммами кооперации, а в наиболее ранних из них — диаграммами взаимодействия.

Диаграмма коммуникации, с одной стороны, обеспечивает концептуально согласованный переход от статической модели диаграммы классов к динамическим моделям поведения, представляемым диаграммами последовательности, состояний и деятельности. С другой стороны, диаграмма этого типа предопределяет особенности реализации модели на диаграммах компонентов и развертывания, которые являются предметом рассмотрения последующих глав книги.

Графической нотацией для представления диаграмм коммуникации, также как и других диаграмм взаимодействия, является фрейм в форме прямоугольника (см. рис. 2.14). В верхнем левом углу фрейма изображается небольшой пятиугольник, в который помещается ключевое слово, или *тег*, **sd**. За этим тегом следует имя фрейма, в данном случае — имя диаграммы коммуникации. Внутри фрейма изображаются линии жизни, связи и сообщения.

В спецификации языка UML 2.0 указано, что все диаграммы, изображающие взаимодействия, следует помечать тегом **sd**. Это представляется спорным в отношении вспомогательных диаграмм, на что обращают внимание многие ведущие эксперты. Так, например, "три амиго" для диаграмм коммуникации рекомендуют использовать тег **comm**. Тем не менее окончательный выбор тега для вспомогательных диаграмм взаимодействия остается за читателем, что не исключает применение собственного варианта.

На диаграмме коммуникации прежде всего изображаются участвующие во взаимодействии линии жизни, особенности представления которых рассматриваются в следующем разделе.

9.1.1. Линия жизни

Линия жизни уже рассматривалась ранее в *главе 7*. В контексте коммуникации линией жизни называют отдельного участника взаимодействия, которым является, как правило, экземпляр класса. На диаграмме коммуникации линия жизни изображается в форме прямоугольника класса (рис. 9.1, *а*). Допускается вместо прямоугольника изображать специальный символ классификатора, который представляет эта линия жизни.

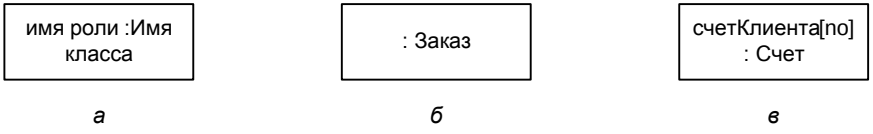


Рис. 9.1. Графическое изображение линий жизни на диаграмме коммуникации: *а* — общая нотация; *б* — пример без имени роли; *в* — пример полного имени с селектором

Информация, идентифицирующая линию жизни на диаграмме коммуникации, изображается внутри прямоугольника в следующем формате (БНФ):

```
<идентификатор-линии-жизни>::= ([<имя-роли> ['['<селектор>']'] ]
[:<имя-класса>]
```

Поскольку аналогичный синтаксис используется для линий жизни на диаграмме последовательности *(см. разд. 7.2)*, во избежание повторений здесь он не рассматривается. В идентификаторе линии жизни можно опустить либо имя роли, либо имя класса. Если имя роли не указано, то, согласно приведенному синтаксису, перед именем класса необходимо указать двоеточие.

Если <имя-роли> опущено, то такую линию жизни часто называют анонимной (рис. 9.1, *б*). Для идентификации конкретной линии жизни, которая представляет отдельный экземпляр класса, используется селектор (рис. 9.1, *в*).

На диаграмме коммуникации, представляющей взаимодействие системы, изображаются только необходимые для моделирования взаимодействия линии жизни, связи и сообщения. Если линия жизни не показана, предполагается, что она не используется в этой модели. Таким образом, на диаграмме коммуникации не обязательно изображать экземпляры всех классов модели и структурных отношений между ними.

Одна и та же совокупность линий жизни может участвовать в реализации различных взаимодействий. При этом, в зависимости от рассматриваемого взаимодействия, могут изменяться как связи между отдельными линиями жизни, так и потоки сообщений между ними. Эта особенность отличает диаграмму коммуникации от диаграммы классов, на которой должны быть ука-

заны все классы, их атрибуты и операции, а также все ассоциации и другие структурные отношения между элементами модели.

На диаграмме коммуникации, которая представляет модель реализации некоторой операции, могут быть показаны символы ролей для аргументов этой операции, и локальные переменные, созданные в ходе ее выполнения. Экземпляры классов, создаваемые или уничтожаемые во время выполнения операции, могут указываться посредством дополнительных примечаний. Предполагается, что экземпляры, не имеющие подобных примечаний, существуют в момент начала выполнения операции и продолжают существовать по окончании ее выполнения. При этом не следует, однако, полагать, что в реализации операции задействуются все линии жизни, изображенные на диаграмме коммуникации.

На диаграммах коммуникации допускается изображать стрелку навигации на линии связи, если эта связь представляет однонаправленный маршрут коммуникации. Однако эта нотация является необязательной, поскольку отсутствует в явном виде в спецификации языка UML 2.0.

На диаграмме коммуникации, как и на диаграмме классов, могут быть показаны структурные отношения между линиями жизни в форме связей.

9.1.2. Связь

Связь (link) является экземпляром произвольной ассоциации, которая обеспечивает канал для направленной передачи сообщений между линиями жизни.

Как правило, связь на диаграмме коммуникации соединяет две линии жизни. Бинарная связь на диаграмме коммуникации изображается отрезком сплошной линии, соединяющей два прямоугольника линии жизни. На каждом из концов этой линии дополнительно могут быть явно указаны имена ролей соответствующей ассоциации.

На диаграмме коммуникации могут быть изображены также рефлексивные связи, которые обеспечивают передачу сообщений линии жизни самой себе. Важно помнить, что в то время как класс требует полного описания всех своих отношений, линия жизни или участник взаимодействия требует описания только тех связей, которые необходимы для реализации отдельного поведения.

Как правило, связи не имеют собственных имен, поскольку в качестве экземпляров ассоциации являются полностью идентичными. Другими словами, все связи на диаграмме коммуникации могут быть только анонимными. Однако чаще всего имена связей или соответствующих ассоциаций на диаграммах

коммуникации не указываются. Для связей не указывается также кратность концевых точек. Однако другие обозначения специальных случаев отношений, такие как агрегация и композиция, могут присутствовать на отдельных концах связей.

Как было отмечено ранее, цель моделирования взаимодействия на диаграмме коммуникации заключается в том, чтобы специфицировать коммуникацию между изображенными линиями жизни. При этом каждое взаимодействие описывается совокупностью сообщений, которыми линии жизни обмениваются между собой.

9.1.3. Сообщение

Сообщения уже рассматривались ранее применительно к диаграммам последовательности в *главе 7*. Напомним некоторые основные положения, которые характеризуют семантику этого понятия.

Сообщение на диаграмме коммуникации специфицирует отдельную единицу коммуникации между двумя линиями жизни, одна из которых передает, а другая принимает от первой некоторую информацию. При этом передающая линия жизни предполагает, что после получения сообщения принимающей линией жизни последует выполнение некоторого действия или операции. Тем самым сообщения не только передают некоторую информацию, но и требуют от принимающей линии жизни выполнения ожидаемых действий.

Сообщение представляет собой законченный фрагмент информации, который специфицирует требования по выполнению действий принимающей линией жизни. Сообщения могут инициировать выполнение операций экземпляром соответствующего класса, а параметры этих операций могут передаваться вместе с сообщением. Таким образом, сообщение является причиной или стимулом для начала выполнения операций, отправки сигналов, создания и уничтожения отдельных экземпляров классов.

Сообщение изображается в форме символа стрелки рядом с линией связи, которое передается в указанном стрелкой направлении по данной связи. Говорят, что стрелка сообщения специфицирует направление коммуникации. Рядом со стрелкой указывается идентификатор сообщения, записанный в специальном формате, синтаксис которого рассматривается в следующем разделе.

В качестве простого примера можно привести фрагмент коммуникации с двумя линиями жизни и одним сообщением (рис. 9.2). Сообщение с именем `оплатить()` имеет направление от анонимного экземпляра класса `Заказ`, который инициирует и отправляет это сообщение, к линии жизни с именем

`счетКлиента` с дополнительным селектором, который идентифицирует экземпляр класса `Клиент` по его номеру.

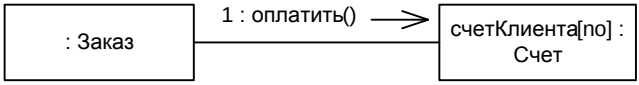


Рис. 9.2. Пример изображения сообщения на диаграмме коммуникации

На диаграмме коммуникации могут использоваться стрелки сообщения различной формы, аналогично рассмотренным ранее стрелкам сообщений на диаграммах последовательности (см. рис. 7.3). Все сообщения на диаграмме коммуникации упорядочены относительно времени своей передачи в соответствии с некоторым уникальным номером.

Семантика сообщения может быть уточнена посредством его вида, который может быть специфицирован формой соответствующей стрелки или специальным текстовым стереотипом. На диаграммах коммуникации используются две основные разновидности сообщений: передача сигнала и вызов операции.

Сигнал представляет собой асинхронное сообщение, которое используется для вызова некоторого поведения у его получателя в момент его приема без необходимости отправки ответных сообщений. Событие передачи сигнала создает запрос передачи и служит причиной наступления события приема сигнала. Если наступление события приема соответствует некоторому триггеру, определенному в классе экземпляра получателя, то оно инициирует выполнение некоторого поведения. При этом владельцем выполнения результирующего поведения является экземпляр класса получателя.

Детали идентификации поведения, которое должно быть вызвано в результате наступления события приема сигнала, а также механизм, посредством которого данные, отправленные с сигналом, становятся доступными для вызываемого поведения, являются точкой изменения семантики. На практике это означает, что разработчик должен дополнительно специфицировать механизм вызова такого поведения.

Синхронное сообщение вызывает некоторую операцию у экземпляра соответствующего класса и, как правило, требует ответа от приемника обратно передатчику. Событие передачи синхронного сообщения создает запрос вызова операции и служит причиной наступления события вызова в приемнике, которое, в свою очередь, инициирует начало выполнения операции. На диаграмме синхронное сообщение специфицируется именем вызываемой операции и значениями аргументов. Синхронное сообщение должно включать всю

необходимую информацию для идентификации выполнения операции. Когда завершается выполнение операции, эта информация также используется, чтобы направить ответное сообщение вызвавшей ее линии жизни.

На диаграмме коммуникации, описывающей реализацию операции, могут быть изображены символы ролей других экземпляров классов, прямо или косвенно используемых линией жизни получателем для выполнения операции. При этом каждое сообщение моделирует один шаг поведения, которое реализует соответствующую операцию. При этом используется специальный формат записи сообщений, который рассматривается в следующем разделе.

9.1.4. Формат записи сообщений

Каждое сообщение может быть помечено строкой текста, которая имеет следующий синтаксис (БНФ):

```
<идентификатор-сообщения>::= [<предшествующие-сообщения>'/']
<выражение-последовательности> ':' [<атрибут>'=']
<имя-операции-или-сигнала> ['('[<аргумент> [','<аргумент>]* ')']
[':' <возвращаемое-значение>]
```

где:

```
<аргумент> ::= ([<имя-параметра>'='] <значение-аргумента>) |
(<атрибут> '=' <имя-out-параметра> [':' <значение-аргумента>] | ' -'
```

Аналогичный синтаксис используется для идентификации сообщений на диаграмме последовательности *(см. разд. 7.3.1)*. Поэтому далее рассматривается назначение только тех термов, которые являются новыми для сообщений на диаграмме коммуникации.

- `<предшествующие-сообщения>` — разделенные запятыми номера сообщений, после которых следует наклонная черта ("слэш"), например, `3, 4/`. Если список номеров `предшествующих` сообщений пуст, то вся запись, включая наклонную черту, опускается. Если указаны номера `предшествующих` сообщений, то они должны соответствовать номерам других сообщений на этой же диаграмме коммуникации. Смысл указания предшествующих сообщений заключается в том, что данное сообщение не может быть передано, пока не будут переданы своим адресатам все сообщения, номера которых записаны в данном списке.
- `<выражение-последовательности>` — разделенный точками список отдельных термов последовательностей, после которого следует двоеточие. Каждый из термов представляет отдельный уровень процедурной вложенности в форме законченной итерации. Наиболее верхний уровень соответ-

ствует самому левому терму последовательности. Каждый из термов последовательности имеет следующий синтаксис:

[<*целое-число*> | <*имя*>] [<*рекуррентность*>]

- <*целое-число*> указывает на порядковый номер сообщения в процедурной последовательности верхнего уровня. Сообщения, номера которых отличаются на единицу, следуют подряд один за другим. Например, сообщение с номером `3.1.4` может следовать за сообщением с номером `3.1.3` в процедурной последовательности `3.1`. Номера последовательности сообщений с одинаковым префиксом формируют отношение упорядоченности и, соответственно, указывают на предшествующие сообщения. Таким предшествующим сообщением будет сообщение с номером, самая правая цифра которого на единицу меньше, чем у рассматриваемого сообщения. Например, сообщение с номером `3.1.4.6` имеет, в качестве предшествующего, сообщение с номером `3.1.4.5` или с любой дополнительной цифрой после `5`.
- <*имя*> в форме буквы некоторого алфавита используется для спецификации параллельных потоков или нитей управления. Сообщения, которые отличаются только именем, являются параллельными на этом уровне вложенности. На одном уровне вложенности все нити управления эквивалентны в смысле приоритета передачи сообщений. Например, сообщения с выражениями `3.1а` и `3.1б` являются параллельными в процедурной последовательности `3.1`.
- <*рекуррентность*> используется для указания итеративного или условного характера выполнения передачи сообщений. Семантика рекуррентности представляет ноль или больше сообщений, которые должны быть выполнены в зависимости от записанного условия. Возможны два варианта записи рекуррентности:
  - ◊ <*рекуррентность*>`::='*''['`<*предложение-итерация*>`']'` для записи итеративного выполнения соответствующего выражения. Итерация представляет последовательность сообщений одного уровня вложенности. Предложение-итерация может быть опущено, если количество итераций никак не специфицируется. Наиболее часто предложение-итерация записывается на некотором псевдокоде или языке программирования. В языке UML 2.0 формат записи этого предложения не определен. Например, рекуррентность вида `*[i:=1..n]` означает последовательную передачу сообщения с параметром `i`, который изменяется от `1` до некоторого целого числа `n` с шагом `1`.
  - ◊ <*рекуррентность*>`::='['`<*предложение-условие*>`']'` для записи ветвления. Эта форма записи специфицирует некоторое условие для

данного сообщения, передача которого по данной ветви возможна только при истинности этого условия. В общем случае *предложение-условие* (condition-clause) является обычным логическим выражением, которое предназначено для синхронизации отдельных нитей потока управления. Оно записывается в квадратных скобках и может быть опущено, если отсутствует у данного сообщения. Наличие этого условия обеспечивает передачу сообщения только в том случае, если это условие принимает значение "истина". Предложение-условие может быть записано на обычном тексте, псевдокоде или некотором языке программирования. Например, запись `[x>y]` означает, что сообщение по некоторой ветви будет передано только в том случае, если значение `x` больше значения `y`.

Следует заметить, что некоторая ветвь обозначается так же, как и некоторая итерация без звездочки. Ее можно представить как итерацию, ограниченную одним наступлением события. Нотация итерации предполагает, что сообщения в этой итерации должны передаваться и выполняться приемником последовательно. Существует также возможность выполнять их параллельно, для чего используется нотация в форме двух вертикальных линий после звездочки: `*| |`.

Заметим также, что итерация не распространяется на вложенные уровни данного потока или нити. Каждый уровень должен иметь собственное представление для итеративного повторения процедурной последовательности.

Можно привести следующие примеры записи сообщений (без номеров предшествующих сообщений):

- `1.2 [(x>=0)&(x<=255)]: отобразитьНаЭкранеЦвет(x, 0, 0)`
- `3.1 [количество цифр номера = 7]: набратьТелефонныйНомер()`

При разработке диаграмм коммуникации вначале изображаются линии жизни и связи между ними. Далее на диаграмму коммуникации необходимо нанести все сообщения, указав их порядок и другие семантические особенности. В качестве примера диаграммы коммуникации можно рассмотреть вариант модели коммуникации внутренних частей банкомата при снятии наличных (рис. 9.3).

Интерпретация этой модели легко определяется из контекста знакомого всем поведения банкомата для типичного хода событий при снятии наличных со счета клиента.

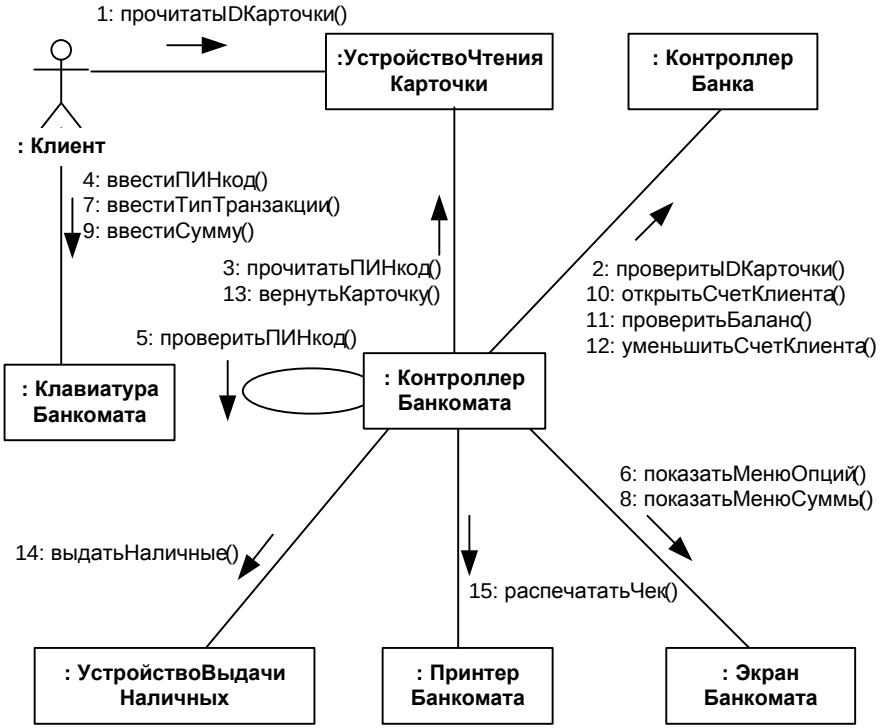


Рис. 9.3. Диаграмма коммуникации для банкомата

9.1.5. Модель коммуникации

Базовая модель коммуникации в языке UML 2.0 по своему замыслу является весьма общей и предусматривает широкий диапазон возможных реализаций. Тем не менее она оказывается полезной на практике, поскольку позволяет задавать основные характеристики коммуникации и требует явно формулировать в модели предположения о логических особенностях передачи сообщений.

Коммуникация между линиями жизни описывается абстрактной моделью передачи сообщений. Процесс коммуникации состоит из следующих действий: `отправка`, `передача`, `вызов` и `ответ`. Получатель сообщения может указываться явным образом в действиях `вызов` и `отправка`, зависеть от контекста выполнения в действии `ответ` или задаваться неявно в среде коммуникации. В случае действий `передача`, `вызов` и `отправка` аргументы действия копируются, и из них формируется сообщение, которое передается одному или нескольким получателям.

При передаче сообщения соблюдается семантика копирования, т. е. аргументы и значения физически не перемещаются, а два сообщения с одинаковыми именами и аргументами не имеют индивидуальности и не различимы между собой. Однако в действительности можно передавать ссылки на аргументы и значения. Точная форма передачи сообщения в языке UML 2.0 не определена, и информация об этой форме может быть недоступна пользовательской программе или доступна в том объеме, в котором хранение информации о реализации предусмотрено специальным профилем. В частности, сообщение может содержать дополнительную информацию о реализации в некоторой среде выполнения.

Способ кодирования сообщений также не задан, что позволяет реализовывать модель коммуникации разными способами. Например, в языке UML 2.0 не делается различия между локальными и удаленными вызовами процедур. Если в реализации требуется маршализация (marshalling) или демаршализация (unmarshalling) аргументов, то заботиться об этом надлежит при написании программного кода, а не в модели коммуникации.

Однако один элемент коммуникации необходимо задавать явно, вне зависимости от реализации. А именно, синхронный вызов операции должен содержать достаточное количество информации о потоке выполнения, инициировавшем этот вызов, чтобы при выполнении действия `ответ` или `возврат` можно было передать ему ответную информацию. Поскольку данная информация невидима для логики приложения, все возможные для принимающей деятельности манипуляции с ней сводятся к посылке ответа. Способ кодирования этой информации существенно зависит от среды выполнения, однако с логической точки зрения он не имеет значения.

Скорость и маршрут передачи сообщения в спецификации UML 2.0 не заданы. Если одновременно передается множество сообщений в форме широковещательной рассылки, то не следует делать каких-либо предположений о порядке их доставки или приема. Если требуется синхронизация, ее необходимо моделировать явным образом. Информация об этом может содержаться в специальном профиле, таком как, например, профиль моделирования систем реального времени.

Обычные вызовы при коммуникации ведут себя как вызовы процедур. Это означает, что в действительности они не требуют каких-либо предварительных действий со стороны принимающей линии жизни. Процесс выполнения создается только после приема вызова. Прием вызова и ресурсы, в рамках которых происходит выполнение, представляют собой детали реализации, но не являются частью логической модели.

Действие `отправка` или `вызов`, активизирующее некоторый переход конечного автомата, должно обрабатываться потоком выполнения принимающей линией жизни. Поток выполнения может моделироваться в виде текущей деятельности, содержащей действия `прием`, или в виде вложенного конечного автомата с переходами. В любом случае сообщения должны каким-то образом собираться и обрабатываться по одному.

Спецификация UML 2.0 обязывает иметь некоторый пул для сбора и хранения событий, включая события передачи и приема сообщений, но не налагает никаких ограничений на способ обработки этих событий, кроме того требования, что они должны обрабатываться по одному. В дополнительных профилях могут быть определены ограничения, типичные для систем реального времени, например, обработка в виде очереди FIFO, приоритеты или временные ограничения на отклик.

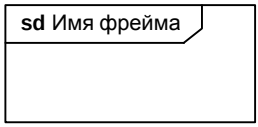
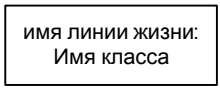
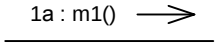
Действие, представляющее вызов характеристики поведения, выполняется передающим объектом, как результат наступления события вызова. Это событие вызова может представлять передачу сигнала или вызов некоторой операции. Как результат данного события вызова генерируется некоторый запрос. Запрос является объектом, захватывающим данные, которые были посланы действию, служащему причиной события вызова. Эта информация может включать в себя: аргументы, которые должны соответствовать параметрам вызываемой характеристика поведения, информацию о природе запроса, т. е. о характеристике поведения, которая должна быть вызвана, особенности передающей и принимающей линий жизни, а также информацию о параметрах вызываемого поведения. При необходимости это должно обеспечить возможность возврата информационно достаточного ответа от вызываемого поведения вызывающей линии жизни.

В то время, как каждое сообщение передается в точности от одной линии жизни и нацелено в точности на одну принимающую линию жизни, сигнал может иметь своим результатом некоторое число сгенерированных запросов, аналогично широковещательной рассылки сигнала.

В модели коммуникации приемник сообщения может быть тем же экземпляром, что и передатчик. Приемник может быть локальным или удаленным. Способ передачи сообщения, величина времени, требуемая для его передачи, порядок, в котором сообщения достигают своих приемников, и физический маршрут достижения принимающих линий жизни в языке UML 2.0 не определены. В модели коммуникации однажды посланное сообщение логически всегда достигает принимающей линии жизни.

В заключение этого раздела приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм коммуникации (табл. 9.1).

Таблица 9.1. *Графические узлы и пути на диаграммах коммуникации*

Тип графического элемента	Нотация	Ссылка на раздел
Фрейм (frame) диаграммы коммуникации		Диаграмма коммуникации *(см. разд. 9.1)*
Линия жизни (lifeline)		Линия жизни *(см. разд. 9.1.1)*
Сообщение (message)		Сообщение *(см. разд. 9.1.3)*

9.2. Диаграмма обзора взаимодействия (interaction overview diagram)

Диаграмма обзора взаимодействия (interaction overview diagram) — диаграмма, которая предназначена для представления взаимодействия только в контексте потока управления в некоторой агрегированной форме.

Диаграммы обзора взаимодействия концентрируют внимание на представлении общего потока управления. Диаграмма коммуникации имеет вид графа, вершинами которого являются взаимодействия или использования взаимодействия, изображенные в виде фреймов. Дугами графа являются направленные линии передачи управления, аналогично диаграммам деятельности. Линии жизни и сообщения на диаграмме этого типа не представляются. Поскольку назначение диаграммы обзора взаимодействия следует также из ее названия, в изображаемом на ней потоке управления каждый узел может быть отдельной диаграммой взаимодействия.

Диаграммы обзора взаимодействия в языке UML 2.0 являются специализациями диаграмм деятельности *(см. главу 8)* и отличаются от них в следующих аспектах:

- диаграммы обзора взаимодействия, вместо узлов действий и объектов диаграмм деятельности, имеют фреймы, каждый из которых может соответствовать взаимодействию или использованию взаимодействия. Эти фреймы

взаимодействия и использования взаимодействия рассматриваются как специальные формы вызовов деятельности;

- альтернативные комбинированные фрагменты представляются узлом решения и соответствующим узлом слияния;
- параллельные комбинированные фрагменты представляются узлом разделения и соответствующим узлом соединения;
- комбинированные фрагменты типа `Цикл` представляются простыми циклами;
- ветвление и слияния ветвлений на диаграммах обзора взаимодействия должны быть должным образом вложенными. Они более ограничены, чем на диаграммах деятельности;
- диаграммы обзора взаимодействия заключаются во фрейм, аналогично другим видам диаграмм взаимодействия. В качестве тега используется уже известное ключевое слово **sd**. Текстовый заголовок фрейма может включать список содержащихся линий жизни, которые графически на диаграмме не представляются.

Диаграммы обзора взаимодействия являются некоторым способом визуализации общей взаимосвязи нескольких взаимодействий на некотором макроуровне, при котором совершенно абстрагируются от сообщений и линий жизни. Поскольку все узлы диаграммы являются фреймами взаимодействия, то и возможности изображения линий жизни и сообщений в этом контексте не существует. Тем не менее линии жизни могут быть изображены внутри фреймов взаимодействия, которые представляются на диаграмме этого типа.

В качестве примера рассматривается диаграмма обзора взаимодействия банкомата (рис. 9.4). Первым выполняется использование взаимодействия `ПредоставитьДоступ` с аргументом `"Карточка действительная"`, после которого следует сообщение с именем `вводПИН-кода()`, изображенное внутри фрейма взаимодействия. Далее существуют две альтернативных ветви управления, которые следуют после первого символа решения в зависимости от правильности введенного ПИН-кода. При правильном ПИН-коде вызывается использование взаимодействия с именем `ПредоставитьМеню` с аргументом `"Опции банкомата"`. В противном случае взаимодействие завершается посредством передачи управления в узел финала деятельности.

Далее также существуют две альтернативных ветви управления, которые следуют после второго символа решения в зависимости от выбранной клиентом опции. Если клиент выбрал снятие наличных, то следует сообщение с именем `вводСуммы()`, изображенное внутри фрейма взаимодействия. Если клиент выбрал получение справки, то вызывается использование взаимодей-

ствия с именем `ПолучитьСправку`. После ввода суммы вызывается использование взаимодействия с именем `ПолучитьНаличные` с аргументом "`Сумма не превышает кредит`". После выполнения одного из указанных взаимодействий обзорное взаимодействие банкомата завершается.

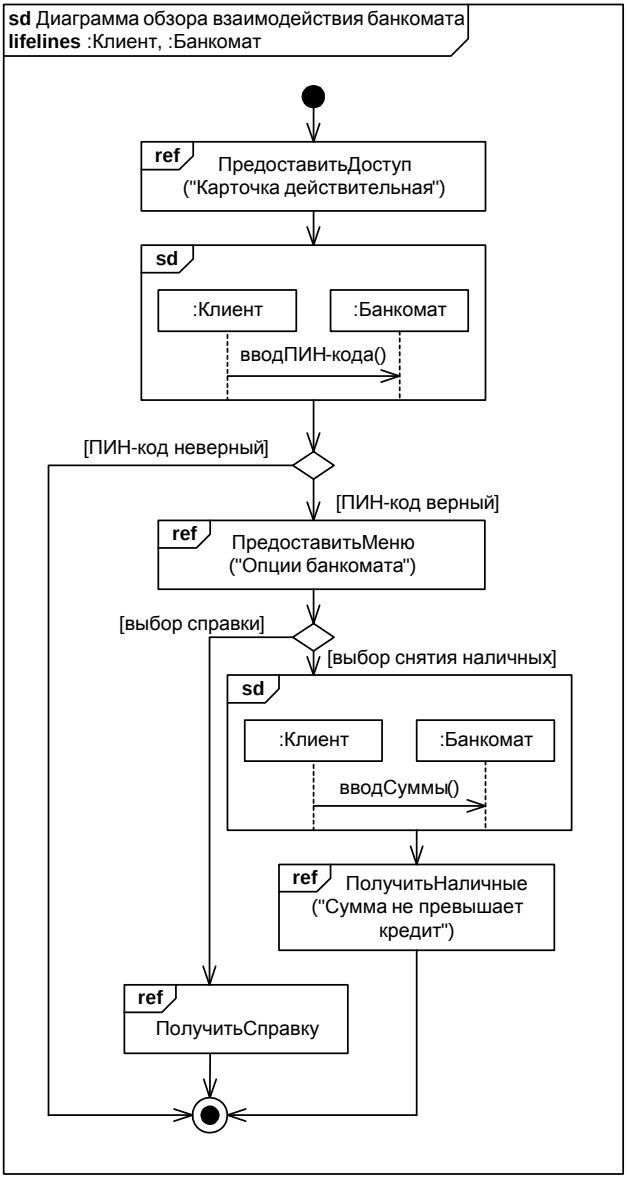


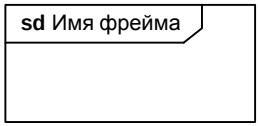
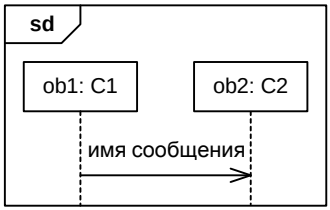
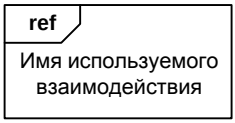
Рис. 9.4. Диаграмма обзора взаимодействия банкомата

Фреймы взаимодействия на диаграмме обзора взаимодействия могут быть или анонимными, или именованными. Фрейм взаимодействия должен быть заменен выполнением соответствующего взаимодействия с заменой его параметров действительными значениями аргументов.

Хотя в спецификации языка UML 2.0 указано, что все диаграммы, изображающие взаимодействия, следует помечать тегом **sd**, "три амиго" для диаграмм обзора взаимодействия рекомендуют использовать тег **intover**.

В заключение этого подраздела приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм обзора взаимодействия (табл. 9.2).

Таблица 9.2. *Графические узлы и пути на диаграммах обзора взаимодействия*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Фрейм (frame)		Диаграмма обзора взаимодействия *(см. разд. 9.2)*
Взаимодействие (interaction)		Диаграмма последовательности *(см. разд. 7.1)*
Использование взаимодействия (interaction use)		Использование взаимодействия *(см. разд. 7.5.1)*

9.3. Временная диаграмма (timing diagram)

Временная диаграмма (timing diagram) — диаграмма взаимодействия, которая служит для представления изменения состояния отдельных линий жизни или особенностей синхронизации сообщений во времени.

Временные диаграммы предназначены для визуализации взаимодействия в контексте времени. В общем случае временная диаграмма используется для отображения изменения в состоянии линии жизни через какое-то время в ответ на внешние события. Линия жизни может представлять экземпляр классификатора или роль классификатора. Временная диаграмма, которую иногда называют диаграммой *синхронизации*, концентрирует внимание на условиях изменения состояния отдельной линии жизни или синхронизации сообщений между несколькими линиями жизни вдоль некоторой линейной оси времени.

Временная диаграмма рассматривается как альтернативное представление диаграммы последовательности, в которой явным образом изображаются изменения состояний линии жизни в заданной шкале времени.

Временные диаграммы в языке UML 2.0 являются специализациями диаграмм последовательности *(см. главу 7)* и отличаются от них в следующих аспектах:

- линейная ось времени обычно изображается таким образом, что текущее время увеличивается в направлении слева направо;
- линии жизни изображаются в отдельных секциях, которые упорядочены по вертикали;
- линия жизни может перемещаться по диаграмме вверх и вниз, что отражает изменение ее состояния. При этом каждому положению по вертикали соответствует отдельное состояние линии жизни. Порядок состояний может иметь некоторый дополнительный смысл;
- линия жизни может быть изображена горизонтально с целью изображения на ней отдельных состояний или значений;
- метрическая ось времени может быть показана с засечками, обозначающими интервалы времени, а в некоторых случаях — дискретные моменты изменений;
- время для всех линий жизни синхронизовано и течет одинаково;
- на диаграмме могут быть показаны значения атрибутов и других свойств линий жизни.

Временные диаграммы заключаются во фрейм, аналогично другим видам диаграмм взаимодействия. В качестве тега используется ключевое слово **sd**.

9.3.1. Основные элементы временной диаграммы

Основные элементы временной диаграммы — линии жизни — на временной диаграмме изображаются в отдельных секциях, слева от которых указываются их имена (рис. 9.5, *а*). На диаграмме также могут быть показаны значения

некоторого атрибута отдельной линии жизни как некоторая функция времени. Значение представляется в форме специального символа (рис. 9.5, *б*) и указывается явно в форме текста. В этом случае пересечение символов значений соответствует событию изменения этого значения. Значения интерпретируются так же, как состояния линии жизни.

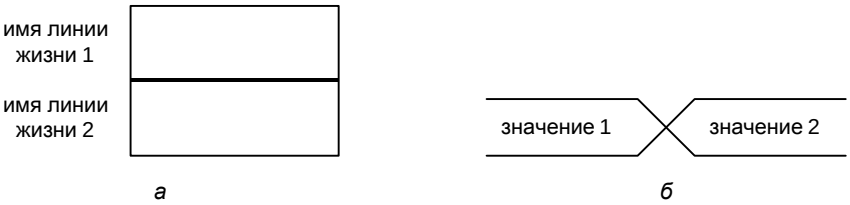


Рис. 9.5. Графическое изображение на временной диаграмме: *а* — линий жизни; *б* — значений

На временных диаграммах могут быть представлены сообщения в различных формах, соответствующие видам сообщений на диаграмме последовательности. Отличие заключается в том, что сообщения изображаются между линиями жизни, которые располагаются вертикально (рис. 9.6, *а*).

Для лучшей визуализации временных диаграмм с большим числом графических элементов сообщение может быть разорвано введением меток с одним и тем же именем (рис. 9.6, *б*). Метки являются всего лишь графическим сокращением, используемым в том случае, когда соединяемые сообщением линии жизни располагаются далеко друг от друга. На диаграмме могут быть изображены состояния классификатора или изменяемые значения атрибута в форме вертикального списка (рис. 9.6, *в*). Обычно эта нотация используется для отображения изменения в состоянии линии жизни через какое-то время в ответ на внешние события или сигналы.

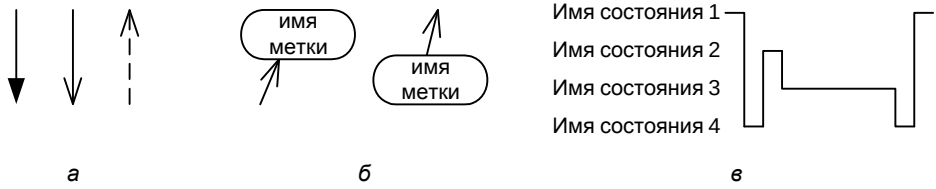


Рис. 9.6. Графическое изображение на временной диаграмме: *а* — сообщений; *б* — меток; *в* — состояний

На временных диаграммах допускается использовать как дискретное, так и непрерывное измерение состояния. В последнем случае это может быть

дополнительной иллюстрацией сценариев, в которых определенные сущности подвергаются непрерывным изменениям состояния, как, например, температура или плотность.

Временные диаграммы могут представлять поведение как индивидуальных классификаторов, так и нескольких взаимодействующих классификаторов. При этом внимание акцентируется на времени наступления событий, которые являются причиной изменения состояний моделируемых линий жизни. Временные диаграммы показывают изменение состояния или значения некоторого атрибута во времени, что особенно актуально в приложениях реального времени. Также как и на диаграммах последовательности, на временных диаграммах могут быть изображены наблюдение времени и временные ограничения.

Существует несколько форм временных диаграмм, наибольшее распространение получили три из них.

9.3.2. Первая форма временной диаграммы

Первая форма используется для представления изменения состояний во времени для отдельной линии жизни. При этом изображаются события, которые являются причиной изменения состояний.

В качестве примера рассмотрим модель доступа сотрудника в рабочее помещение, на входе в которое установлена специальная система контроля. Для доступа необходимо вставить специальную карту в считывающее устройство и набрать индивидуальный код. Временная диаграмма может иллюстрировать изменение состояний линии жизни `Сотрудник` при входе в рабочее помещение (рис. 9.7). На ней изображены 4 состояния этой линии жизни с именами: `Ожидание`, `Ожидание Карты`, `Ожидание Доступа` и `Проход`.

Данная диаграмма может быть интерпретирована следующим образом: `Сотрудник` находится в исходном состоянии `Ожидание`, которое изменяется в момент ввода кода. Это отражается указанием соответствующего события или стимула рядом со "скачком" изменения. После ввода кода `Сотрудник` переходит в состояние `Ожидание Карты`, в котором может находиться не более 3-х интервалов продолжительности, величина которых соответствует наблюдению продолжительности `d`. На это указывает специальное ограничение продолжительности `{d..3*d}`.

Изменение состояния `Ожидание Карты` происходит в момент возврата карты. При этом `Сотрудник` переходит в состояние `Ожидание Доступа`. В этом состоянии он находится не более 3-х стандартных единиц времени, которое отсчитывается от некоторого момента `t`. На это указывает специальное временное ограничение `{t..t+3}`. После этого `Сотрудник` переходит в состояние `Проход`, что соответствует его входу в рабочее помещение.

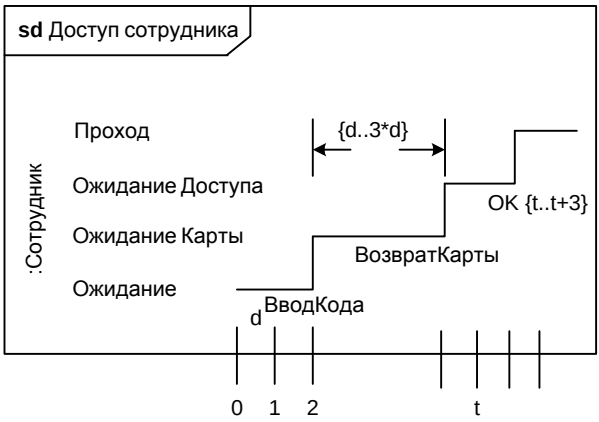


Рис. 9.7. Пример первой формы временной диаграммы модели доступа сотрудника в помещение

9.3.3. Вторая форма временной диаграммы

Вторая форма временной диаграммы также используется для представления изменения состояний для отдельной линии жизни горизонтально во времени. При этом события, которые являются причиной изменения состояний, не изображаются. Это наиболее компактная форма временной диаграммы, которая часто используется для иллюстрации процесса функционирования электронных устройств, таких как процессор, модуль памяти, отдельные шины и магистрали материнской платы, различные аппаратные интерфейсы.

Рассматриваемый пример временной диаграммы второго типа иллюстрирует только изменение состояний линии жизни `Сотрудник` при входе в рабочее помещение без указания событий (рис. 9.8).

Данная диаграмма может быть легко интерпретирована по аналогии с предыдущим примером.

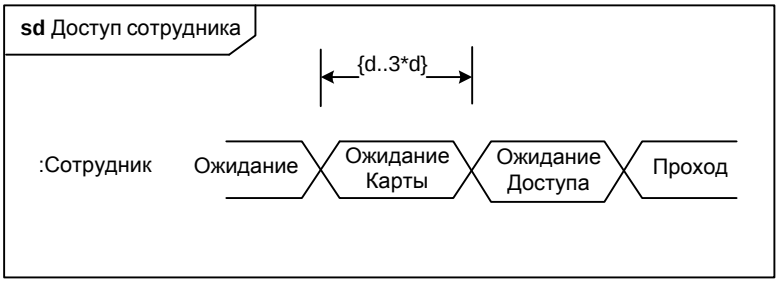


Рис. 9.8. Пример второй формы временной диаграммы модели доступа сотрудника в помещение

9.3.4. Третья форма временной диаграммы

Третья форма временных диаграмм является наиболее продвинутой, поскольку на ней могут быть изображены более чем одна линия жизни и сообщения между этими линиями жизни. Эта форма временной диаграммы по своим возможностям в большой степени соответствует диаграмме последовательности, чем первые две формы.

Рассматриваемый пример временной диаграммы третьего типа иллюстрирует не только изменение состояний линия жизни `Сотрудник`, но и изменение состояний линия жизни `Система Контроля`. При этом дополнительно указаны 5 сообщений, передаваемых между этими линиями жизни (рис. 9.9). Интересно отметить, что сообщение с именем `отпереть_Дверь` изображено нацеленным на границу фрейма диаграммы, что соответствует его передаче во внешний шлюз. Интерпретация других элементов данной диаграммы аналогична ранее рассмотренным примерам.

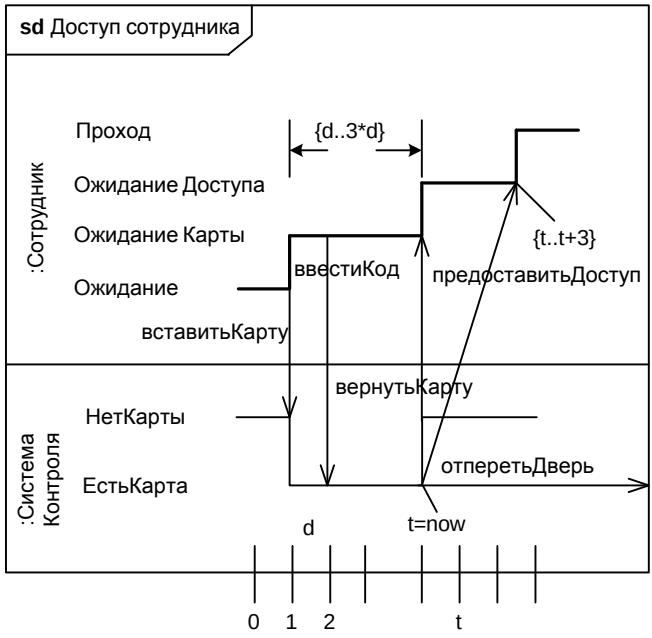


Рис. 9.9. Пример третьей формы временной диаграммы модели доступа сотрудника в помещение

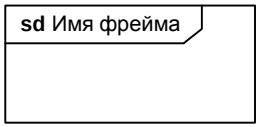
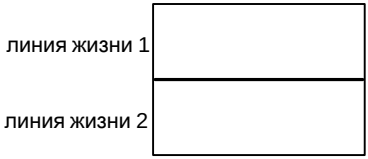
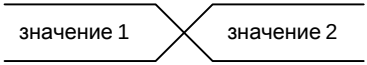
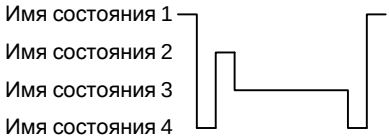
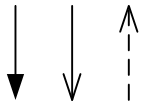
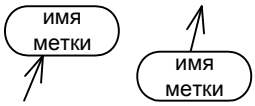
Как нетрудно заметить, эта форма временной диаграммы позволяет наглядно представить синхронизацию передаваемых сообщений и соответствующее

изменение состояний нескольких линий жизни, что имеет важное значение при моделировании систем реального времени или, как их часто называют, реактивных систем.

* * *

В заключение этого раздела приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении временных диаграмм (табл. 9.3).

Таблица 9.3. *Графические узлы и пути на временных диаграммах*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Фрейм (frame)		Временная диаграмма *(см. разд. 9.3)*
Линия жизни (lifeline)		Основные элементы временной диаграммы *(см. разд. 9.3.1)*
Общие значения линии жизни (general value lifeline) или состояния (states)		Основные элементы временной диаграммы *(см. разд. 9.3.1)*
Линия времени состояния (state timeline) или условия (condition timeline)		Основные элементы временной диаграммы *(см. разд. 9.3.1)*
Сообщение (message)		Основные элементы временной диаграммы *(см. разд. 9.3.1)*
Метка сообщения (message label)		Основные элементы временной диаграммы *(см. разд. 9.3.1)*
Событие разрушения (destruction event)		Линия жизни *(см. разд. 7.2)*

Глава 10

Диаграмма конечного автомата (state machine diagram)

В этой главе описываются графические элементы, которые могут быть изображены на диаграммах конечного автомата, а также приводится конкретная информация о семантике и нотации каждого элемента. В ней также рассматриваются примеры, которые иллюстрируют, как эти графические элементы используются на диаграммах конечного автомата.

Диаграмма конечного автомата (state machine diagram) является графом, который представляет некоторый конечный автомат.

Диаграмма конечного автомата может изображаться в форме фрейма или без него. Если для диаграммы используется нотация фрейма, то имя диаграммы записывается в секции имени фрейма, аналогично другим диаграммам языка UML 2.0.

Вершинами диаграммы или графа конечного автомата являются символы состояния. Кроме символов обычных состояний на диаграмме конечного автомата могут быть изображены другие типы вершин, которые получили специальное название *псевдосостояний*. Граф конечного автомата является ориентированным. Это означает, что состояния на диаграмме конечного автомата соединяются между собой направленными дугами. Каждая такая дуга называется *переходом*. В общем случае вершина в графе конечного автомата может быть источником или назначением любого количества переходов.

Для понимания особенностей моделирования и графического представления конечных автоматов в языке UML 2.0 необходимо рассмотреть формализм конечного автомата, который является объектно-ориентированным расширением диаграмм состояний, предложенных Д. Харелом (D. Harel).

10.1. Концептуальные основы моделирования конечных автоматов в языке UML 2.0

Конечный автомат (state machine) представляет собой некоторый формализм для моделирования поведения отдельных элементов модели или системы в целом.

Концепция конечного автомата определяет множество понятий, которые могут быть использованы для моделирования дискретного поведения в форме графических моделей типа "состояние — переход". Конечные автоматы могут быть использованы для спецификации поведения различных элементов модели. Например, конечный автомат может использоваться для моделирования поведения индивидуальной сущности, такой как экземпляр класса. Конечные автоматы могут специфицировать поведение совокупности элементов некоторой системы, например, экземпляра композитного класса или кооперации. Так, например, поведение кооперации представляет поведение реализации всех ее частей, а поведение класса представляет поведение только экземпляров одного класса.

Кроме моделирования поведения, конечные автоматы могут применяться также для спецификации протоколов использования системы. В связи с этим в языке UML 2.0 рассматриваются два типа конечных автоматов: конечные автоматы поведения и протокольные конечные автоматы. Из них наибольшее применение на практике нашли конечные автоматы поведения.

Поведение (behavior) является спецификацией того, как экземпляр классификатора изменяет значения отдельных характеристик в течение своего времени жизни.

В общем случае конечные автоматы поведения могут находиться в собственности классификатора и определяться в контексте этого классификатора. Это позволяет конечным автоматам поведения ссылаться на характеристики поведения и другие свойства этого классификатора. Поведение классификатора, моделируемое конечным автоматом, вызывается при создании экземпляра этого классификатора. В этом случае параметры конечного автомата соответствуют параметрам характеристики поведения.

При вызове поведения классификатора используется некоторое множество входных значений, которые оказывают влияние на ход его выполнения. По окончании своего выполнения поведение в качестве результата производит

множество выходных значений, которые соответствуют его параметрам. При выполнении поведения могут создаваться и разрушаться объекты, изменяться значения их атрибутов, генерироваться сообщения между объектами, а также рассчитываться значения выходных параметров деятельностей.

Поведение моделируется посредством последовательного прохождения вершин графа конечного автомата, которые соответствуют отдельным состояниям.

Состояние (state) — элемент модели поведения, предназначенный для представления ситуации, в ходе которой поддерживается некоторое условие инварианта.

Инвариант состояния специфицирует условия, которые всегда должны быть истинными, если данное состояние является текущим состоянием. Инвариант состояния может представлять статическую ситуацию, в ходе которой, например, объект ожидает наступления некоторого внешнего события. Инвариант состояния может также моделировать динамические условия, такие как характеристики элемента модели при входе и выходе из состояния. В языке UML 2.0 различают следующие виды состояний: простое состояние, композитное состояние и состояние подавтомата. Далее каждая из этих разновидностей состояний рассматривается более подробно.

Вершины графа конечного автомата соединяются между собой дугами переходов.

Переход (transition) представляет собой направленное отношение между двумя состояниями, одно из которых является *вершиной-источником (source vertex)*, а другое — *целевой вершиной (target vertex)*.

В ходе выполнения поведения состояние конечного автомата может быть активным или неактивным. Состояние становится активным, если в него происходит вход в результате срабатывания некоторого перехода, и становится неактивным, если из него происходит выход в результате срабатывания некоторого перехода. В частном случае вход в состояние и выход из состояния может быть в результате срабатывания одного и того же перехода.

В ходе срабатывания перехода могут быть выполнены некоторые действия. Если такое действие является вызовом синхронной операции у объекта, то этот переход не закончит своего срабатывания до тех пор, пока вызванный объект не завершит выполнение своей операции.

Переходы конечного автомата срабатывают посредством наступления или совершения некоторых событий.

Событие (event) является спецификацией некоторых условий, которые оказывают влияние на поведение моделируемой сущности.

Поведение всегда выполняется в контексте своего объекта, независимо и параллельно с выполнением любого другого поведения. В этом случае объект, который является контекстом поведения, управляет специальным входным пулом. Последний предназначен для обработки событий, на которые данное поведение может реагировать. Если объект имеет более одного ассоциированного поведения, то все поведения должны иметь доступ к этому пулу событий. При этом объект гарантирует, что каждое наступление события во входном пуле может быть поглощено только одним поведением.

Часто события происходят в результате выполнения некоторого действия в системе или в ее окружении. При наступлении таких событий они размещаются во входном пуле объекта. После этого событие берется из входного пула и направляется поведению классификатора принимающего объекта для обработки. В этой точке считается, что событие поглощается из пула, и на него ссылаются, как на текущее событие. В дальнейшем поглощенное событие недоступно для обработки.

Какие бы то ни было предположения относительно интервалов времени между наступлением события, правила диспетчеризации событий в пуле и их поглощения в общем случае отсутствуют. Это оставляет открытой возможность различных семантических вариаций, таких как предположение о мгновенном характере наступления события. Семантической вариацией является также ситуация, перестает ли использоваться наступившее событие, если для него в конечном автомате не существует подходящего триггера.

В прежних версиях языка UML термин "событие" использовался некорректно, поскольку он соответствовал как типу, так и экземпляру события. В языке UML 2.0 термин "событие" означает тип, а термин *"наступление события"* (occurrence) — экземпляр. При необходимости можно использовать фразы "тип события" для события и "наступление события" для экземпляра события. Следует заметить, что это согласуется с часто используемым в русском и английском языках фразы "событие наступило", которая предполагает наступление события определенного типа. Так, например, при описании ситуации времени выполнения можно сказать "событие X наступило" или "наступило событие X". С другой стороны, фраза "произошло наступление события для заданного типа события" выглядит избыточной и неуклюжей.

Событие, которое может являться причиной выполнения некоторого ассоциированного поведения, специфицируется посредством триггера.

Триггер (trigger) устанавливает отношение события с поведением, которое может оказывать влияние на экземпляр классификатора.

Триггер специфицирует событие, которое может вызывать выполнение поведения, а также любые ограничения на это событие с целью фильтрации событий, которые не представляют интереса. Триггер используется для определения некоторого безымянного события. Формальный синтаксис для триггера определяется посредством явного указания подкласса события следующим образом:

<*триггер*>::= <*событие-вызова-операции*>|<*событие-сигнала*>|
<*событие-любого-приема*>|<*временное-событие*>|<*событие-изменения*>

Когда объект, являющийся экземпляром классификатора с поведением, распознает наступление некоторого события, то или это может иметь непосредственный эффект, или наступление события может быть сохранено во входном пуле объекта. Непосредственный эффект проявляется либо в виде срабатывания перехода, либо в вызове специфицированного поведения. Хранение наступления события во входном пуле событий объекта осуществляется до тех пор, пока он не будет использован некоторым триггером. Потребление наступления события происходит в тот момент времени, когда тип наступившего события в пуле совпадет с некоторым триггером. В этой точке вызывается ассоциированное поведение или срабатывает переход, как это определено данным событием.

Когда поведение, которое выполняется в контексте объекта, подходит к точке, где триггер должен продолжить его выполнение, проверяется входной пул для события, которое удовлетворяет одному или нескольким невыполненным триггерам. Если событие удовлетворяет одному из триггеров, то это событие удаляется из входного пула, и поведение продолжает свое специфицированное выполнение. Любые данные, ассоциированные с этим событием, становятся доступными в этот момент времени.

События из входного пула объекта направляются на обработку и обрабатываются конечным автоматом по одному в каждый момент времени. Порядок выбора из очереди не определен, оставляя открытой возможность моделирования различных схем приоритета. Семантика обработки наступления событий основывается на предположении *выполнения до завершения*. Обработка выполнения до завершения означает, что наступление события может быть взято из входного пула объекта и направлено на обработку только в том случае, если обработка предыдущего текущего наступления события полностью завершена.

Обработка одного наступления события конечным автоматом называется *шагом выполнения до завершения*. Перед началом шага выполнения до за-

вершения конечный автомат находится в некоторой стабильной конфигурации состояний со всеми завершенными входными, выходными и внутренними действиями, но не обязательно завершенной do-деятельностью. Такие же условия должны быть выполнены после окончания шага выполнения до завершения. Таким образом, наступление события никогда не будет обрабатываться в то время, когда конечный автомат находится в некоторой промежуточной и неустойчивой ситуации. Шаг выполнения до завершения является переходом между двумя конфигурациями состояний конечного автомата.

Предположение выполнения до завершения упрощает функцию перехода конечного автомата, поскольку в ходе обработки события исключаются конфликты параллельности, позволяя конечному автомату безопасно завершить свой шаг выполнения до завершения. Когда обнаруживается наступление события и направляется на обработку, это может привести к одному или более переходам, разрешенным для срабатывания. Если же никакой переход не срабатывает, и это событие (точнее — его тип) не входит в список отложенных событий текущей конфигурации состояний, то это наступление события перестает использоваться, и шаг выполнения до завершения заканчивается.

Любое вызываемое поведение может, в свою очередь, вызывать другие поведения, которые определены в контексте этого же классификатора. Конечный автомат владеет одним или более регионами, которые, в свою очередь, владеют соответствующими вершинами и переходами. При наличии ортогональных регионов в результате одного и того же наступления события возможно срабатывание нескольких переходов — не менее одного перехода в каждом регионе в текущей конфигурации состояний.

В случае, когда разрешены один или более переходов, конечный автомат выбирает некоторое подмножество и запускает их. Какие из разрешенных переходы в действительности сработают — определяется согласно алгоритму выбора перехода, который описывается далее в этой главе. Порядок, в котором выбранные переходы сработают, для общего случая не определен и может быть основан на некотором явном или неявном приоритете.

На практике информация о наступлении событий может зависеть от вида выполняемых действий, свойств среды коммуникации и целого ряда других факторов. В одних случаях наступление события происходит мгновенно и без потери информации, в то время как в других случаях оно может происходить с задержкой, с потерей данных, с нарушением порядка и с дублированием. Концепция конечного автомата абстрагируется от подобных практических аспектов поведения, что служит точкой изменения семантики в языке UML 2.0.

Контекст классификатора с поведением, моделируемым конечным автоматом, определяет, какие триггеры и вызовы операций определены для конечного

автомата, а также какие атрибуты и операции являются доступными при вызове соответствующего поведения. Триггеры сигналов и триггеры вызовов операций для конечного автомата определяются в соответствии с операциями этого классификатора. Как вид поведения, конечный автомат может иметь некоторую ассоциированную характеристику поведения или являться методом этой характеристики поведения. В этом случае конечный автомат специфицирует эту характеристику поведения классификатора.

Конечный автомат без контекста классификатора может использовать триггеры, которые являются независимыми от получателей. В этом случае они являются триггерами сигналов или триггерами вызовов операций, основанных на параметрах шаблона операции параметризованного конечного автомата.

10.2. Простое состояние

Простым состоянием (simple state) называется состояние, которое не имеет внутренних регионов и подсостояний.

Простое состояние изображается прямоугольником с закругленными углами с именем состояния, изображенным внутри этого прямоугольника (рис. 10.1, *а*). Имя состояния допускается также указывать во вкладке имени, которая изображается в форме небольшого прямоугольника, присоединенного к верхней стороне символа состояния (рис. 10.1, *б*). Символ простого состояния может быть разделен также на несколько секций, отделенных друг от друга горизонтальными линиями (рис. 10.1, *в*).

Секциями простого состояния являются: секция имени, секция внутренней деятельности, секция внутренних переходов. Каждая из этих секций описывается далее.

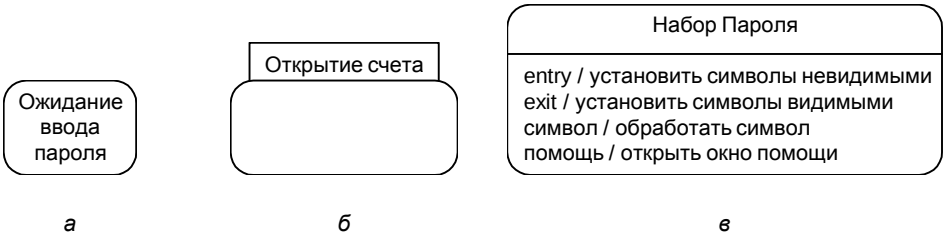


Рис. 10.1. Варианты изображения простого состояния:
а — в виде прямоугольника с закругленными углами;
б — в виде прямоугольника со вкладкой, содержащей имя состояния;
в — в виде прямоугольника с разделением на несколько секций

10.2.1. Секция имени

Эта секция изображается в верхней части символа состояния и содержит необязательное имя состояния в форме строки текста, которая начинается с прописной буквы. На диаграмме конечного автомата допускается изображать состояния без имени, которые называются *анонимными* состояниями. Все анонимные состояния считаются различными. На одной диаграмме нежелательно показывать одно и то же именованное состояние дважды, поскольку это может привести к недоразумению. Секция имени не должна использоваться, если применяется вкладка имени (см. рис. 10.1, *б*), и наоборот.

Имена состояний могут быть квалифицированы посредством указания имен всех композитных состояний и регионов, в которых они содержатся. В этом случае квалифицированное имя данного состояния должно быть записано в следующей форме: "`Регион-или-Состояние-А :: Регион-или-Состояние-Б :: Данное-Состояние`". В этой записи предполагается, что `Регион-или-Состояние-А` непосредственно содержит `Регион-или-Состояние-Б`, которое непосредственно содержит `Данное-Состояние`. Квалифицированные имена состояний могут быть использованы в тех случаях, когда состояния с одним и тем же именем находятся в различных регионах или композитных состояниях.

10.2.2. Секция внутренней деятельности

Эта секция содержит список внутренних действий или деятельность, которые выполняются во время нахождения элемента в данном состоянии. Для спецификации этих действий предназначена метка деятельности, которая идентифицирует обстоятельства выполнения этих действий или деятельности. После метки деятельности в форме строки текста записывается выражение деятельности, которое отделяется от метки знаком `/` (обратный слэш). Если выражение деятельности является пустым, то разделитель `/` необязателен.

В языке UML 2.0 могут использоваться следующие метки деятельности.

- `entry`. Эта метка специфицирует поведение, которое также называют входным поведением. Это поведение выполняется всякий раз, когда происходит вход в данное состояние независимо от перехода, позволившего достичь это состояние. Спецификация входного поведения является необязательной и называется также *действиями на входе*. Если для некоторого состояния определены действия на входе, то они всегда выполняются до своего полного завершения и до начала выполнения любого внутреннего поведения или срабатывания переходов, которые могут быть обработаны в данном состоянии.

- `exit`. Эта метка специфицирует поведение, которое также называют выходным поведением. Это поведение выполняется всякий раз, когда происходит выход из данного состояния независимо от перехода, который выводит из этого состояния. Спецификация выходного поведения является необязательной и называется также *действиями на выходе*. Если для некоторого состояния определены действия на выходе, то они всегда выполняются до своего полного завершения только после того, как все внутренние деятельности и действия переходов завершат свое выполнение.
- `do`. Эта метка специфицирует поведение, которое выполняется до тех пор, пока моделируемый элемент находится в данном состоянии, или до тех пор, пока не закончится выполнение деятельности, специфицированной соответствующим выражением. В последнем случае дополнительно может быть сгенерировано событие завершения. Спецификация этого поведения является необязательной и называется также *"ду-деятельностью"*. Выполнение "ду-деятельности" начинается, когда происходит вход в состояние, и прекращается либо само по себе, либо когда произойдет выход из этого состояния, какой бы вариант ни был первым. В случае, когда существует некоторый выходящий из состояния переход по завершении "ду-деятельности", произойдет выход из этого состояния. При выходе из состояния "ду-деятельность" должна быть завершена до того, как будут выполнены возможные действия на выходе. Если состояние должно быть покинуто в результате срабатывания триггерного перехода, а "ду-деятельность" еще не завершена, то эта "ду-деятельность" прерывается, и перед выходом из состояния выполняются только возможные действия на выходе.

Рассмотренные метки деятельности не могут быть использованы в качестве имен событий или переходов.

В качестве примера действия на входе можно привести действие "установить символы невидимыми", действия на выходе — действие "установить символы видимыми", внутренних переходов — `символ / обработать символ` и `помощь / открыть окно помощи` (см. рис. 10.1, *в*).

10.2.3. Секция внутренних переходов

Эта секция содержит список внутренних переходов, каждый из которых специфицируется отдельной строкой текста в соответствии с рассмотренным ранее формальным синтаксисом для триггера. *Внутренний* (internal) переход выполняется без выхода из состояния и без повторного входа в состояние, в котором он определен. Имена внутренних переходов в одном состоянии

могут повторяться только в том случае, если для них указываются различные сторожевые условия. Параметры события и сторожевые условия являются необязательными.

Таким образом, внутренние переходы не являются причиной изменения состояния. Это означает, что входные или выходные действия состояния источника не будут вызываться. Внутренний переход может сработать, даже если конечный автомат находится в одном или более регионах, вложенных в состояние-источник.

10.2.4. Отложенные события

Для состояния дополнительно может быть специфицировано множество типов событий, которые являются *отложенными* (defer) в этом состоянии. Событие, которое не инициирует никаких переходов в текущем состоянии, не может быть обработано. Однако если его тип совпадает с одним из типов в множестве отложенных событий этого состояния, то оно остается в пуле событий. Отложенные события сохраняются до тех пор, пока не будет достигнуто другое состояние, в котором эти события смогут инициировать некоторый переход.

На диаграмме конечного автомата отложенные события часто изображаются в секции внутренней деятельности. При этом каждое отложенное событие представляется отдельной строкой текста в соответствии с рассмотренным ранее формальным синтаксисом для триггера, после которой следует разделитель `/` и ключевое слово `defer`.

Кроме обычных состояний на диаграмме конечного автомата используется целый ряд вспомогательных состояний, которые имеют "проходной" характер и называются псевдосостояниями.

10.3. Псевдосостояния

Псевдосостояние (pseudo state) — абстрактный элемент модели, который включает в себя различные типы вспомогательных вершин в графе конечного автомата.

Псевдосостояние называют также *транзиентной* вершиной графа конечного автомата, понимая под этим тот факт, что в любом из псевдосостояний моделируемое поведение не задерживается, а следует далее. Вид псевдосостояния является перечислением следующих значений литералов: начальное (initial), завершение (terminate), выбор (choice), соединение (junction), разделение (fork), слияние (join), точка входа (entry point), точка выхода (exit point),

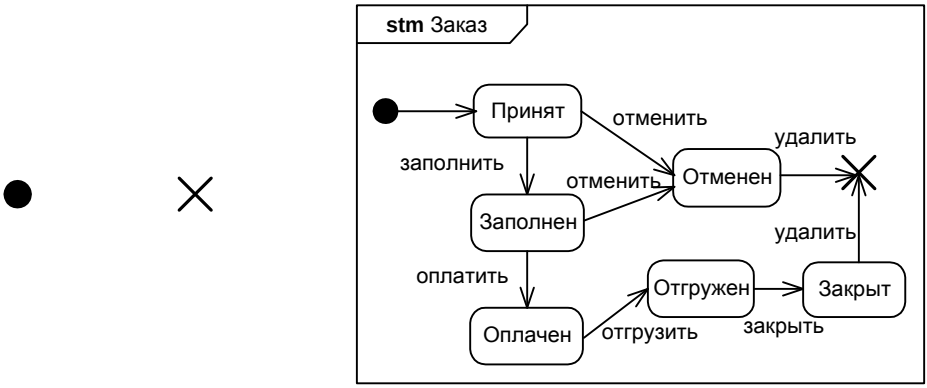
неглубокая история (shallow history), глубокая история (deep history). Далее каждый из этих видов псевдосостояний рассматривается более подробно.

Формально псевдосостояния не могут иметь входного и выходного поведения, а также выполняющегося поведения ("ду-деятельности"). Что касается имени псевдосостояний, то в языке UML 2.0 это никак не специфицировано.

10.3.1. Начальное псевдосостояние

Начальное псевдосостояние (initial pseudo state) представляет вершину графа конечного автомата, которая по умолчанию является состоянием-источником для начального перехода моделируемого поведения.

Начальное псевдосостояние, которое также называют *начальной вершиной*, *начальным узлом* или просто *начальным состоянием*, может иметь только один выходящий переход, который также называют *начальным переходом*. На диаграмме конечного автомата или в любом регионе не может быть более одной начальной вершины. Начальное псевдосостояние изображается в форме небольшого круга (рис. 10.2, *а*).



а *б* *в*

Рис. 10.2. Начальное псевдосостояние (*а*), узел завершения (*б*) и пример их использования (*в*)

Переход из начальной вершины может быть помечен событием только в рассматриваемых далее регионах конечного автомата. Это событие соответствует созданию объекта, в контексте которого рассматривается моделируемое

поведение. Во всех остальных случаях начальный переход должен быть непомеченным и не может содержать сторожевого условия. Однако начальный переход может иметь необязательное действие на переходе. Начальный переход на самом верхнем уровне или регионе конечного автомата либо не имеет триггера, либо он имеет триггер со стереотипом `<<create>>`.

10.3.2. Узел завершения

Узел завершения (terminate node) является псевдосостоянием, вход в который означает завершение выполнения поведения конечного автомата в контексте его объекта.

Вход в узел завершения эквивалентен вызову действия разрушения объекта. Узел завершения изображается в форме перекрестия, аналогично символу разрушения объекта (рис. 10.2, *б*).

В качестве примера использования начальной вершины и узла завершения рассмотрим диаграмму конечного автомата для экземпляра класса `Заказ` (рис. 10.2, *в*). Эта диаграмма, изображенная в форме фрейма, описывает жизненный цикл любого отдельного заказа. При поступлении заказа от клиента создается отдельный экземпляр класса `Заказ`, что соответствует переходу из начальной вершины в состояние `Принят`. Далее при заполнении заказа наступает соответствующее событие, и объект переходит в состояние `Заполнен`. Если же происходит отмена заказа, то объект переходит в состояние `Отменен`. Аналогично интерпретируются переходы и в другие состояния. В узел завершения можно перейти в результате удаления заказа либо из состояния `Отменен`, либо из состояния `Закрыт`. В этом узле выполняется разрушение соответствующего экземпляра и, если не предусмотрено заранее, вся информация об этом экземпляре теряется.

10.3.3. Выбор

Псевдосостояние выбора (choice pseudo state) предназначено для моделирования нескольких альтернативных ветвей при реализации поведения конечного автомата.

Псевдосостояние выбора, которое также называют *вершиной* или *узлом выбора*, или просто — *выбором*, позволяет расщепить входящий в него переход на несколько выходящих путей. Выбор ветви происходит на основе динамического оценивания сторожевых условий триггеров для каждого из выходящих переходов. Решение, какой путь следует принять, может быть функцией результатов предшествующих действий, осуществленных на этом же шаге

выполнения до завершения. Выходящие из выбора переходы не должны иметь пометок имен событий.

Узел выбора изображается в форме символа ромба (рис. 10.3). Каждый выходящий из узла выбора переход должен иметь обязательное сторожевое условие, которое представляет собой логическое условие, записанное в квадратных скобках (рис. 10.3, *а*).

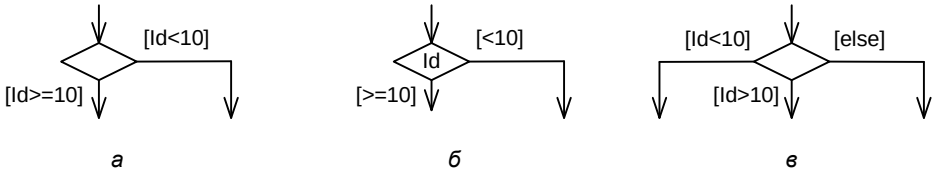


Рис. 10.3. Варианты изображения узла выбора: *а* — с двумя сторожевыми условиями; *б* — с общим левым операндом; *в* — с тремя сторожевыми условиями

Если для всех переходов, выходящих из некоторого узла выбора, сторожевые условия являются бинарными выражениями, которые совместно используют общий левый операнд, то нотация для узла выбора может быть упрощена. Этот левый операнд может быть помещен внутри символа ромба, а остальные части выражения сторожевых условий помещаются на выходящих переходах (рис. 10.3, *б*).

Эта альтернативная нотация для выбора является источником многочисленных ошибок, которые на практике допускают некоторые разработчики, помещая в символ ромба вместо левого операнда само логическое условие. Для пущей убедительности они добавляют в конце условия знак вопроса ?, а выходящие переходы помечают словами "да" и "нет". Поскольку подобная нотация отсутствует в языке UML 2.0, то соответствующую модель следует признать несогласованной (ill-formed) или, другими словами, содержащей ошибки в графической нотации.

В языке UML 2.0 на диаграммах конечного автомата допускается изображать узлы выбора с выходящими переходами, для которых несколько сторожевых условий могут быть истинными одновременно. Если для узла выбора более чем одно из сторожевых условий оценивается как "истина", то для продолжения все равно должен быть выбран один путь. Каким образом это реализуется на практике — остается вопросом, который должен уточнить сам разработчик.

Если в момент проверки логического выражения для узла выбора ни одно из сторожевых условий выходящих переходов не может быть оценено как "истина", то модель считается несогласованной (ill-formed). Чтобы избежать этого, разработчикам рекомендуется для узлов выбора с нетривиальными логическими условиями определять один выходящий переход со сторожевым условием `[else]` (рис. 10.3, *в*).

Узлы выбора следует отличать от узлов соединения, которые рассматриваются далее.

10.3.4. Соединение

Псевдосостояние соединения (junction pseudo state) является вершиной со свободной семантикой, которая используется для соединения вместе нескольких переходов.

Псевдосостояние соединения, которое также называют *вершиной* или *узлом соединения*, или просто — *соединением*, обычно используется для соединения нескольких переходов. Вершина соединения должна иметь по крайней мере два входящих перехода и по крайней мере один выходящий переход. Наиболее часто соединение используется для сведения воедино несколько входящих переходов в один выходящий переход. Выходящие из соединения переходы не должны иметь пометок имен событий.

Узлы соединения могут быть также использованы для расщепления некоторого входящего перехода на несколько выходящих переходов с различными сторожевыми условиями. Эта конструкция реализует статические альтернативные ветви, которые отличаются от динамических условных ветвлений, которые моделируются посредством рассмотренных ранее вершин выбора.

Узел соединения изображается небольшим черным кругом, по размерам несколько меньшим, чем начальная вершина (рис. 10.4, *а*).

Узлы соединения могут использоваться для построения сложных путей переходов. Так, например, объединив несколько переходов, входящих в узел разделения, и множество переходов, выходящих из узла разделения, можно получить некоторый составной переход, обладающий дополнительными свойствами. При этом как входящие, так и выходящие переходы могут иметь сторожевые условия (рис. 10.4, *б*).

Допускается для выходящих переходов использовать предварительно определенное сторожевое условие, обозначаемое через `[else]`. Однако это сторожевое условие может иметь не более чем один выходящий переход. Этот

переход активизируется, если все сторожевые условия, помечающие другие выходящие переходы, принимают значение "ложь".

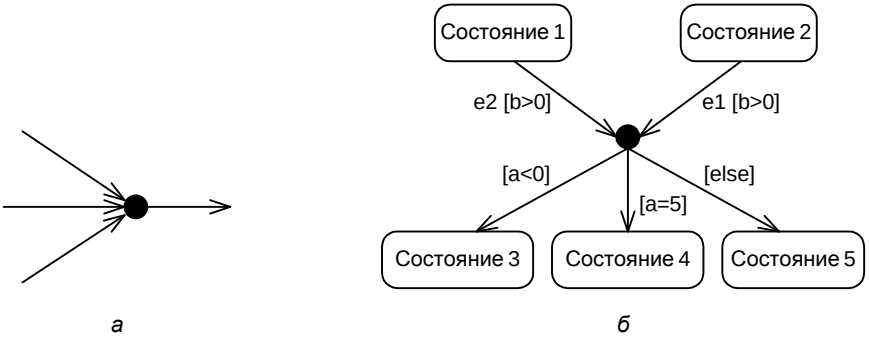


Рис. 10.4. Варианты изображения узла соединения: *а* — общий вид; *б* — пример использования

Статический характер проверки сторожевых условий для узла соединения с последующим выбором условной ветви проявляется, в частности, в том, что выходящие переходы, чьи сторожевые условия оцениваются как значение "ложь", могут вообще не изображаться на диаграмме. Это отличает данную конструкцию от узла выбора. Так, например, если заменить в рассмотренном фрагменте (см. рис. 10.4, *б*) сторожевое условие `[else]` на `[a>7]`, то диаграмма будет являться согласованной (well formed). Наоборот, подобное изменение для узла выбора привело бы к несогласованной (ill-formed) диаграмме.

10.3.5. Разделение

Вершина разделения (fork vertex) является псевдосостоянием, предназначенным для разделения входящего перехода на два или более перехода, которые имеют в качестве своих целей вершины в ортогональных регионах композитного состояния.

Вершина разделения, которую также называют *узлом разделения* или просто — *разделением*, позволяет расщепить входящий в него переход на несколько выходящих параллельных путей. В конечном автомате вершина разделения должна иметь ровно один входящий переход и по крайней мере два выходящих перехода. Все переходы, выходящие из вершины разделения, должны иметь целевые состояния в различных ортогональных регионах некоторого композитного состояния. Понятия ортогонального региона и ком-

позитного состояния рассматриваются далее в этой главе. Выходящие из разделения переходы не должны иметь сторожевые условия или триггеры.

Узел разделения изображается в форме короткой утолщенной черты (рис. 10.5, *слева*), из которой выходят две или более стрелки переходов.

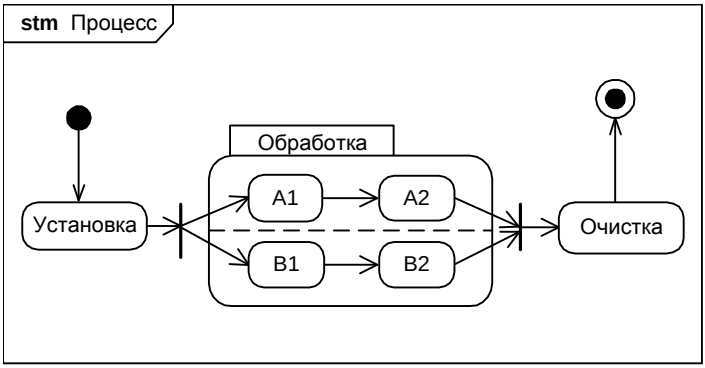


Рис. 10.5. Пример изображения узлов разделения *(слева)* и слияния *(справа)*

Как правило, каждому узлу разделения на диаграмме конечного автомата должен соответствовать узел слияния.

10.3.6. Слияние

Вершина слияния (join vertex) является псевдосостоянием, предназначенным для соединения нескольких переходов, которые имеют в качестве своих источников вершины из различных ортогональных регионов композитного состояния.

Переходы, входящие в вершину слияния, не могут иметь сторожевые условия или триггеры. Все переходы, входящие в вершину слияния, должны брать начало в различных регионах некоторого композитного состояния.

Нотацией для разделения и соединения является короткая утолщенная черта (см. рис. 10.5, *справа*), в которую входят две или более стрелки переходов.

Диаграмма конечного автомата (рис. 10.5) содержит так называемые нетриггерные переходы или переходы по завершению "ду-деятельности". Поэтому для всех состояний необходимо специфицировать соответствующие внутренние деятельности. Заметим также, что на этой диаграмме изображено и финальное состояние, которое рассматривается далее.

10.3.7. Точка входа

Точка входа (entry point) является псевдосостоянием, предназначенным для моделирования входа в некоторый конечный автомат или композитное состояние.

Точка входа используется для соединения некоторого внешнего перехода, оканчивающегося в этой точке входа с некоторым внутренним переходом, исходящим из этой точки входа. Точка входа может быть определена только на самых верхних регионах конечного автомата. Конечный автомат или композитное состояние может иметь не более одного выходящего из точки входа перехода в вершину в этом же регионе. Точка входа изображается в форме небольшой окружности на границе диаграммы конечного автомата или композитного состояния (рис. 10.6, *а*). Точка входа может иметь необязательное имя, которое изображается в форме строки текста со строчной буквы рядом с ее символом.

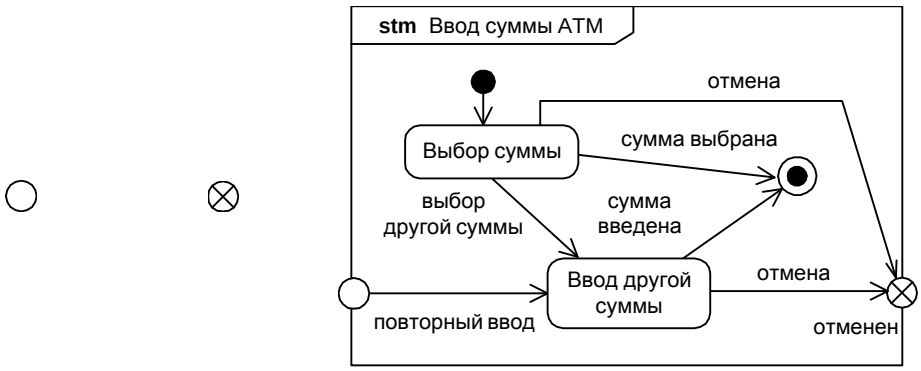


Рис. 10.6. Точка входа (*а*), точка выхода (*б*) и пример их использования (*в*)

10.3.8. Точка выхода

Точка выхода (exit point) является псевдосостоянием, предназначенным для моделирования выхода из некоторого конечного автомата или композитного состояния.

Точка выхода используется для соединения некоторого внутреннего перехода, оканчивающегося в этой точке выхода с некоторым внешним переходом,

исходящим из этой точки выхода. Точка выхода может быть определена только на самых верхних регионах конечного автомата. Попадание в точку выхода означает выход из соответствующего конечного автомата или композитного состояния. Точка выхода изображается в форме небольшой окружности с перекрестием на границе диаграммы конечного автомата (см. рис. 10.6, *б*). Точка выхода может иметь необязательное имя, которое изображается рядом с ее символом.

В качестве примера использования точек входа и выхода можно рассмотреть диаграмму конечного автомата для ввода суммы при снятии наличных в банкомате (см. рис. 10.6, *в*). Точка входа с именем `повторный ввод` может быть использована для повторного ввода суммы, а точка выхода с именем `отменен` — для моделирования отмены действий клиента при вводе суммы. Следует заметить, что входящие в точку выхода переходы могут быть вообще не изображены на диаграмме, если выходящий из конечного автомата переход имеет некоторое имя триггера.

Символы точки входа и точки выхода могут быть помещены также внутри или за пределами границы диаграммы конечного автомата или композитного состояния. Эти варианты представления рассматриваются в языке UML 2.0 как дополнительные.

10.3.9. Неглубокая история

Две разновидности исторических псевдосостояний используются только внутри композитного состояния. Хотя все особенности моделирования композитных состояний рассматриваются далее в этой главе, тем не менее, описание состояний неглубокой и глубокой истории приводятся в этом разделе совместно с другими псевдосостояниями.

Псевдосостояние неглубокой истории (shallow pseudo state) предназначено для представления самого последнего активного подсостояния композитного состояния после выхода из него.

Псевдосостояние неглубокой истории, которое называют также *вершиной* или *узлом неглубокой истории*, или просто — *неглубокой историей*, указывает на последнее активное подсостояние, но не на вложенные в него другие подсостояния. Таким образом, переход, входящий в вершину неглубокой истории, эквивалентен переходу, входящему в наиболее последнее активное подсостояние композитного состояния. Если для композитного состояния специфицированы входные действия, то они выполняются при входе в композитное состояние перед переходом из вершины неглубокой истории.

Вершина неглубокой истории изображается в форме небольшого круга, содержащего букву H (рис. 10.7). Она изображается внутри композитного состояния, которое непосредственно его содержит. При этом композитное состояние не может иметь более одной вершины неглубокой истории на одном уровне вложенности. Из вершины неглубокой истории должен выходить только один переход. Этот переход принимается во внимание в том случае, если рассматриваемое композитное состояние никогда ранее не было активным.

В качестве примера использования неглубокой истории рассмотрим композитное состояние `Редактирование текста` (рис. 10.7). При первоначальном входе в это состояние выполняется входное действие по переводу редактора в активный режим, после чего осуществляется переход в состояние `Набор текста`. После окончания набора выполняется переход в состояние `Проверка правописания`. Если в результате проверки выявлены ошибки, то происходит переход в состояние `Внесение исправлений`. Если же ошибки отсутствуют, то выполняется переход в финальное состояние, которое рассматривается далее в этом разделе. Финальное состояние завершает редактирование текста, после чего выполняется перевод редактора в пассивный режим и переход рассматриваемой модели в другое состояние, которое в этом фрагменте не показано.

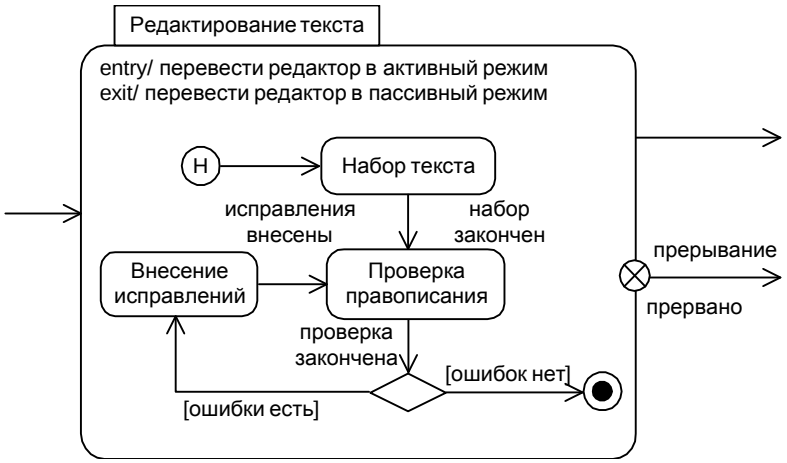


Рис. 10.7. Пример использования псевдосостояния неглубокой истории

Выход из композитного состояния также может произойти в результате прерывания, которое моделируется посредством перехода с именем `прерывание` и точки выхода с именем `прервано`. В эту точку выхода произойдет переход из любого подсостояния композитного состояния `Редактирование текста` при наступлении прерывания. Перед этим будет выполнен перевод редактора

в пассивный режим, после чего триггерный переход направит рассматриваемую модель в другое состояние, которое на этом фрагменте не показано. Неглубокая история запомнит последнее активное состояние на момент прерывания. При следующем входе в это композитное состояние, возможно — после обработки прерывания, неглубокая история направит переход в то состояние, которое было в нем сохранено.

Следует заметить, что выход из композитного состояния по финальному состоянию "обнуляет" неглубокую историю или, другими словами, она теряет свою историю. Поэтому при повторном входе в композитное состояние, если ранее был выполнен из него последний выход по финальному состоянию, вершина неглубокой истории задействует переход по умолчанию, например, переход в подсостояние `Набор текста` (см. рис. 10.7).

10.3.10. Глубокая история

Псевдосостояние глубокой истории (deep pseudo state) предназначено для представления последней активной конфигурации композитного состояния после выхода из него.

Псевдосостояние глубокой истории, которое называют также *вершиной* или *узлом глубокой истории*, или просто — *глубокой историей*, указывает на последнее активное подсостояние, включая также все его активные подсостояния любого уровня вложенности. Таким образом, переход, входящий в вершину глубокой истории, эквивалентен переходу, входящему в наиболее последнее активное подсостояние самого глубокого уровня вложенности для композитного состояния, которое ее содержит. Если для вложенных композитных состояний специфицированы входные действия, то все они выполняются при входе в композитное состояние перед переходом из вершины глубокой истории.

Вершина глубокой истории изображается в форме небольшого круга, содержащего букву H со звездочкой * (рис. 10.8). Она изображается внутри композитного состояния, которое непосредственно его содержит. При этом композитное состояние не может иметь более одной вершины глубокой истории. Из вершины глубокой истории должен выходить только один переход. Этот переход принимается во внимание в том случае, если рассматриваемое композитное состояние никогда ранее не было активным.

В качестве примера использования глубокой истории рассмотрим уточненный фрагмент композитного состояния `Редактирование текста`, которое, в свою очередь, также содержит композитное состояние `Набор текста` (см. рис. 10.8). При первоначальном входе в композитное состояние `Редактирование`

текста выполняется входное действие по переводу редактора в активный режим, после чего осуществляется переход в композитное состояние Набор текста. После входа в это композитное состояние происходит переход в подсостояние Ввод символов, после окончания ввода выполняется переход в состояние Форматирование. Затем происходит переход в подсостояние Проверка правописания и далее — как уже было рассмотрено ранее в аналогичном примере.

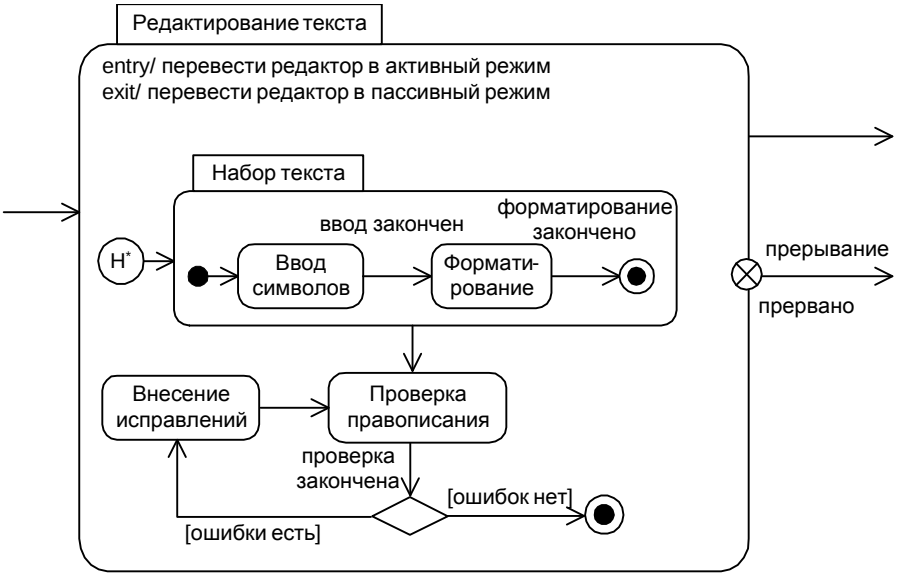


Рис. 10.8. Пример использования псевдосостояния глубокой истории

Выход из любого композитного состояния может произойти в результате прерывания посредством точки выхода, которая изображена на границе первого из них и оказывает свое влияние на любое вложенное в него композитное состояние. При наступлении прерывания в эту точку выхода произойдет переход из любого вложенного подсостояния композитного состояния Редактирование текста, включая и подсостояния композитного состояния Набор текста. Глубокая история запомнит конфигурацию всех последних активных подсостояний на момент прерывания. При следующем входе в композитное состояние Редактирование текста глубокая история направит переход в подсостояние той конфигурации, которая была в нем сохранена.

Так, например, если в момент наступления прерывания активным было подсостояние Форматирование, то глубокая история запомнит его и композитное

состояние `Набор текста`. При следующем входе в композитное состояние `Редактирование текста` глубокая история сразу направит переход в подсостояние `Форматирование`.

Следует заметить, что выход из композитного состояния по финальному состоянию также "обнуляет" и глубокую историю одного с ним уровня вложенности, посредством чего глубокая история теряет свою историю. Поэтому при повторном входе в композитное состояние, если ранее был выполнен из него последний выход по финальному состоянию, вершина глубокой истории задействует переход по умолчанию, например, переход в подсостояние `Набор текста` (см. рис. 10.8). Создается впечатление, что глубокая история эквивалентна нескольким неглубоким историям, по одному на каждом уровне вложенности. Однако в спецификации языка UML 2.0 об этом ничего не говорится.

10.3.11. Финальное состояние

Финальное состояние (final state) — специальный вид состояния, предназначенного для моделирования завершения конечного автомата или региона, в котором оно содержится.

Строго говоря, финальное состояние не является псевдосостоянием. Когда происходит вход в финальное состояние, содержащий его конечный автомат или регион завершается. Завершение конечного автомата означает прекращение поведения объекта, который представляет контекст этого конечного автомата. Поэтому финальное состояние не может иметь никаких выходящих переходов. Финальное состояние изображается в форме окружности, внутри которой помещен небольшой черный круг (на рис. 10.5—10.7, 10.8, *справа*).

Если регион, в котором достигнуто финальное состояние, непосредственно содержится в некотором конечном автомате, и все другие регионы в этом конечном автомате также завершены, то это означает, что весь конечный автомат завершается. Если регион содержится в конечном автомате, то переход из этого региона по его завершении в другое состояние или регион конечного автомата может иметь в качестве нотации только непомеченный переход.

10.4. Переход

Общее определение перехода и его нотация и простейшие свойства были рассмотрены ранее в *разд. 1.1*. Отдельные свойства переходов были также описаны при рассмотрении псевдосостояний в *разд. 1.3*. Настоящий раздел посвящен детальному рассмотрению семантики и синтаксису записи имени различных видов переходов в конечном автомате.

В общем случае переход может быть помечен строкой текста, синтаксис которой определяется следующим выражением (БНФ):

<*переход*>:: = <*триггер*> [','<*триггер*>]* ['['<*сторожевое-ограничение*>']'] ['/'<*выражение-деятельности*>]

Здесь <*триггер*> представляет собой имя триггерного события, вернее, его типа. В качестве имени триггера могут использоваться имена операций с параметрами или без них. Терм <*сторожевое-ограничение*>, которое в прежних версиях языка UML 1.*x* называется *сторожевым условием* (guard condition), является логическим выражением, записанным в терминах параметров триггерного события, атрибутов и связей объекта контекста. Терм <*выражение-деятельности*>, который иногда называют *выражением поведения*, выполняется, если и только если переход срабатывает.

Имя перехода может быть записано в терминах операций и атрибутов, а также связей объекта контекста и параметров триггерного события. При этом может использоваться имя любой другой характеристики, видимой в области контекста объекта поведения. Выражение поведения может содержать список, состоящий из нескольких различных действий. Это могут быть действия, которые явно генерируют события, такие как передача сигналов или вызовы операций.

Приведенное выше выражение для синтаксиса записи имени перехода (БНФ), которое взято из спецификации языка UML 2.0, строго говоря, представляется не совсем корректным. Дело в том, что эта запись справедлива только для записи имени триггерных переходов. Что касается нетриггерных переходов, то для них имя триггера не является необходимым, поскольку они происходят по завершении "ду-деятельности". По этой причине первый терм рассматриваемого выражения должен быть заключен в квадратные скобки: [<*триггер*>], что делает его необязательным.

10.4.1. Сторожевое условие

Сторожевое условие (guard condition) предназначено для более детального управления срабатыванием перехода. В простом переходе с одним сторожевым условием оно оценивается до срабатывания этого перехода. Если сторожевое условие в момент оценивания принимает значение "истина", то переход может сработать, в противном случае он не срабатывает.

Хотя в спецификации языка UML 2.0 термин "сторожевое условие" заменен на термин "ограничение", но, строго говоря, записывается оно, как и в прежних версиях языка UML 1.*x*, а именно — в квадратных скобках. Поскольку для записи ограничений во всех версиях языка UML используются фигурные скобки, это решение вряд ли следует признать удачным. Более того, ограничение для ассоциированного с ним элемента модели всегда должно принимать значение "истина". Сторожевое условие может принимать также и значение "ложь".

Именно по этим причинам автор оставляет в книге для сторожевого ограничения прежнее название — сторожевое условие, тем более, что он не усматривает в этом никакого семантического эффекта. Другие варианты перевода английского guard типа *охрана* и пр. представляются автору менее удачными. Кстати, в оригинале и переводе книги "трех амиго", посвященной описанию языка UML 2.0, также используется термин "сторожевое условие" (guard condition).

В рассматриваемых далее составных переходах, которые могут содержать несколько сторожевых условий, все сторожевые условия оцениваются до срабатывания этого перехода, за исключением, когда в составном переходе используется узел выбора. Если в составном переходе существует узел выбора, то согласно рассмотренному выше правилу оцениваются только те сторожевые условия, которые предшествуют этому узлу выбора. Сторожевые условия, находящиеся после некоторой точки выбора, оцениваются после того, когда эта точка выбора будет достигнута. Другими словами, для оценивания значения сторожевого условия точка выбора имеет тот же самый эффект, что и состояние.

Пример записи перехода с триггером, сторожевым условием и действием на переходе: `нажатьПравуюКнопкуМыши (координаты) [координаты находятся в активном окне] / выделитьОбъект()`.

Сторожевые условия не должны включать выражения, приводящие к побочным эффектам. Модели, которые нарушают это, рассматриваются как несогласованные (ill formed).

10.4.2. Переходы завершения и события завершения

Переход завершения является переходом, который выходит из состояния или точки выхода и не имеет явного триггера, хотя он может иметь сторожевое условие. Переход завершения неявно срабатывает посредством некоторого события завершения. В частном случае событие завершения может быть сгенерировано после того, как будут выполнены входные действия и внутренние деятельности ("ду-деятельности"). Если для события никаких входных действий или деятельностей не существует, то событие завершения генерируется при входе в это состояние.

Если состояние является рассматриваемым далее композитным состоянием или состоянием подавтомата, то событие завершения генерируется, если соответствующий подавтомат или регион достигнет финального состояния, а внутренние деятельности в этом состоянии будут завершены. Это событие

является неявным триггером для перехода завершения. Например, переход завершения, выходящий из некоторого ортогонального композитного состояния, должен сработать автоматически, как только все ортогональные регионы достигнут своего финального состояния.

Если для некоторого состояния определено несколько переходов завершения, то они должны иметь взаимно исключающие сторожевые условия. Модели, которые нарушают это, рассматриваются как несогласованные (ill formed).

10.4.3. Составные переходы

Описываемая далее семантика выполнения переходов относится к общему случаю перехода, когда он является составным.

Составной переход (compound transition) является производным семантическим понятием, которое представляет "семантически полный" путь, совершаемый одним или несколькими переходами.

Составной переход является ациклической непрерывной цепочкой переходов, которые могут быть соединены посредством псевдосостояний слияния, соединения, выбора или разделения. Эти псевдосостояния определяют полный путь из множества состояний источников (возможно — из единственного состояния) в множество целевых состояний или состояний назначения (возможно — в единственное состояние). Для *рефлексивного* перехода одно и то же состояние выступает как в качестве множества источников, так и в качестве множества назначения. Простой (не составной) переход связывает только два состояния и является частным случаем составного перехода.

Начало составного перехода может иметь несколько переходов, выходящих из множества взаимно ортогональных регионов, которые соединяются посредством узла слияния. Конец составного перехода может иметь несколько переходов, выходящих из узла разделения и направленных в множество взаимно ортогональных регионов. В составном переходе из общего узла соединения могут выходить несколько переходов. В этом случае сработать может только один выходящий переход, сторожевое условие которого принимает значение "истина". Если несколько выходящих переходов имеют сторожевые условия со значением "истина", то из этого множества переходов должен быть выбран только один. Алгоритм выбора такого перехода в языке UML 2.0 не специфицирован. Следует заметить, что в этом случае сторожевые условия оцениваются до срабатывания этого составного перехода.

В составном переходе, когда из общего узла выбора выходят несколько переходов, сработает выходящий переход, сторожевое условие которого принимает значение "истина" во время достижения этой точки выбора. Если несколько выходящих переходов имеют сторожевые условия со значениями "истина", то из этого множества переходов должен быть выбран только один. Алгоритм выбора такого перехода в языке UML 2.0 не специфицирован. Детальное описание правил разрешения и срабатывания переходов приводится в *разд. 10.4.7*.

10.4.4. Передача сигнала

Передача сигнала является действием, которое имеет специальную графическую нотацию, описанную ранее при рассмотрении диаграмм деятельности *(см. разд. 8.3.1)*. Напомним, что результатом передачи сигнала может быть начало выполнения некоторой деятельности или срабатывание перехода некоторого конечного автомата.

Узел передачи сигнала обозначается с помощью символа выпуклого пятиугольника, в котором записывается в качестве имени тип сигнала (см. рис. 8.13, *а*). В этом символе могут быть изображены действительные параметры этого сигнала. В составном переходе узел передачи сигнала должен следовать после символа триггера, если последний существует. В общем случае составной переход может иметь несколько узлов передачи сигнала.

10.4.5. Прием сигнала

Прием сигнала, также называемый *приемом триггера*, является действием, которое имеет специальную графическую нотацию, описанную ранее при рассмотрении диаграмм деятельности *(см. разд. 8.3.2)*. Прием сигнала представляет триггер перехода, поскольку приводит к срабатыванию перехода некоторого конечного автомата.

Узел приема сигнала обозначается с помощью символа вогнутого пятиугольника, в котором записывается в качестве имени тип сигнала (см. рис. 8.14, *а*). Текст спецификации триггера помещается в этом символе. Если переход имеет сторожевое условие, то это сторожевое условие также записывается в символе приема сигнала в форме следующего выражения:

```
<прием-сигнала>:: = <триггер> [',' <триггер>]*
['['<сторожевое-условие>']']
```

Символ приема сигнала или триггера является всегда первым в составном переходе. При этом любой составной переход не может иметь более одного такого символа.

В качестве примера рассмотрим упрощенную диаграмму конечного автомата, которая представляет поведение по обработке запросов к базе данных (рис. 10.9). Диаграмма содержит два состояния, которые связывает составной переход, имеющий в качестве триггера узел приема сигнала с именем `запрос` и параметром `id`. Далее следует символ решения с двумя выходящими переходами, каждый со своим сторожевым условием. После этих выходящих переходов следуют узлы передачи сигнала и узлы действия на переходе. Выполнение этого перехода подчиняется правилам разрешения и срабатывания составных переходов, которые рассматриваются далее.

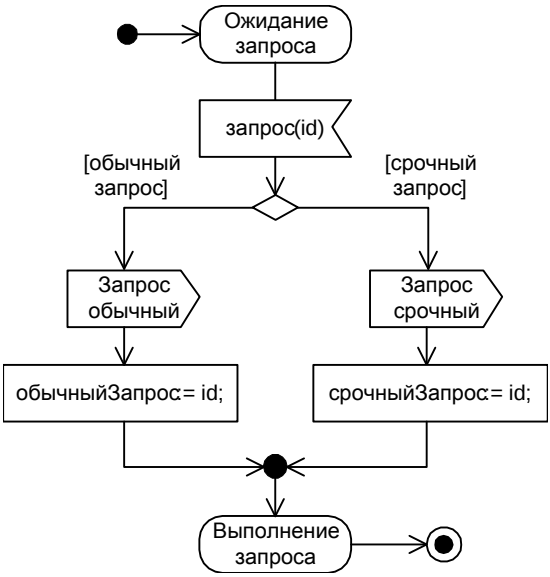


Рис. 10.9. Пример диаграммы конечного автомата с символами приема сигнала, передачи сигнала и действий на переходе

10.4.6. Действия на переходе

На диаграммах конечного автомата могут быть изображены действия, которые выполняются при срабатывании составных переходов. В общем случае действие на переходе специфицируется в форме текста. Альтернативно действия могут быть представлены в графической форме с помощью символа узла действия. Символом узла действия на диаграмме конечного автомата является прямоугольник, внутри которого записывается спецификация этого действия. В составном переходе узел действия должен следовать после узла приема сигнала, если оба они присутствуют в рассматриваемом переходе.

Следует заметить, что узлы выбора и соединения не представляются действиями на переходе, поскольку они являются псевдосостояниями и для них имеются специальные символы выбора и соединения.

В рассмотренной диаграмме конечного автомата (см. рис. 10.9) представлены два узла действия, которые могут быть выполнены при срабатывании составного перехода альтернативно после проверки условия в узле выбора. Каждое из действий присваивает значение идентификатора поступившего запроса атрибуту объекта, который, возможно, является контекстом для моделируемого поведения. Информация об этом на данной диаграмме не специфицирована.

10.4.7. Правила разрешения и срабатывания переходов

В общем случае для составного перехода имеет место следующее правило разрешения. Составной переход считается *разрешенным* (enabled), если и только если:

- все его состояния-источники принадлежат активной конфигурации состояний;
- один из триггеров этого перехода удовлетворяет текущему наступлению события (точнее — типу события). Некоторое событие *удовлетворяет* триггеру, если оно совпадает с событием, специфицированным посредством этого триггера. Некоторое событие приема сигнала удовлетворяет событию сигнала, если оно ассоциировано с тем же сигналом или его обобщением;
- существует, по крайней мере, один полный путь из конфигурации состояний источников либо к конфигурации целевых состояний, либо к некоторому узлу выбора, в котором хотя бы одно сторожевое условие для выходящих переходов принимает значения "истина". При этом переходы без сторожевых условий срабатывают всегда, т. к. их сторожевые условия по умолчанию принимают значения "истина".

Поскольку одно и то же событие может быть причиной разрешения более чем одного перехода, "быть разрешенным" — *необходимое*, но не достаточное условие для срабатывания перехода.

В общем случае для составного перехода имеет место следующее правило определения главного состояния источника и главного целевого состояния.

Каждый переход, за исключением внутренних и локальных переходов, может служить причиной выхода из некоторого состояния источника и входа в целевое состояние. Эти два состояния, которые могут быть композитными, воспринимаются как главный источник и главная цель перехода. Состояние *наименьший общий предок* (Least common ancestor, LCA) некоторого составного перехода является регионом или ортогональным состоянием, которое является LCA состояний источника и цели этого составного перехода. Предполагается, что операция LCA определена для метакласса `Конечный Автомат`. Если LCA является регионом, то главный источник является прямым подсостоянием региона, который содержит эти состояния-источники, а главная цель является подсостоянием региона, который содержит эти целевые состояния. В случае, когда LCA является ортогональным состоянием, главный источник и главная цель представляются посредством самого этого ортогонального состояния.

Причина введения в рассмотрение операции LCA заключается в том, что переход, пересекающий регионы ортогонального композитного состояния, соответствует выходу из этого ортогонального состояния и повторному входу во все его регионы.

Если переход удовлетворяет правилу разрешения, для него определены главное состояние-источник и главное целевое состояние, то он может сработать. В общем случае для составного перехода имеет место правило срабатывания или выполнения, которое представляет собой последовательность выполнения следующих шагов в указанном порядке:

1. Происходит должным образом выход из главного состояния источника.
2. В последовательности переходов выполняются специфицированные действия в соответствии с их линейным порядком вдоль сегментов составного перехода. При этом чем ближе действие к состоянию-источнику, тем более ранним оно выполняется на этом составном переходе.
3. Если встречается некоторый узел выбора, то динамически оцениваются сторожевые условия, следующие после этого узла выбора, и выбирается некоторый путь, сторожевые условия которого принимают значения "истина".
4. Происходит должным образом вход в главное целевое состояние.

Для срабатывания составного перехода необходимо также принимать во внимание свойства рассмотренных ранее псевдосостояний, например, переходы, входящие в вершину слияния, не могут иметь сторожевые условия или триггеры; выходящие из соединения переходы не должны иметь пометок имен событий *(см. разд. 10.3)*.

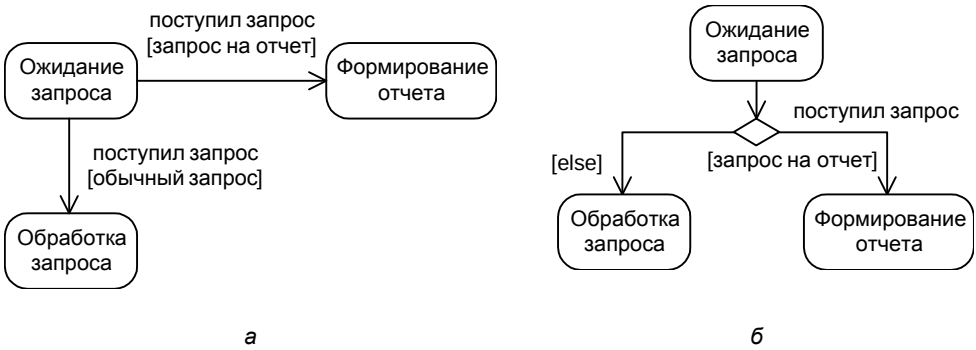
10.4.8. Конфликтующие переходы

Согласно правилам разрешения и срабатывания переходов, в конечном автомате в некоторый момент времени может быть разрешен более чем один переход. Если это имеет место, то такие переходы могут конфликтовать друг с другом.

Два и более разрешенных перехода называются *конфликтующими (conflicting)*, если все они выходят из одного и того же состояния, или, более точно, пересечение множеств их состояний источников не является пустым.

На диаграмме конечного автомата одновременно могут сработать только те переходы, которые разрешены во взаимно ортогональных регионах. Только такая новая активная конфигурация состояний является согласованной (well formed). Внутренний переход в некотором состоянии может конфликтовать только с переходами, которые выходят из этого состояния.

В качестве примера рассмотрим случай двух переходов со сторожевыми условиями, которые имеют в качестве целей простые состояния (рис. 10.10, *а*).



а *б*

Рис. 10.10. Пример конфликта переходов (*а*) и вариант устранения конфликта (*б*)

Пусть эти переходы выходят из одного и того же состояния и разрешены при одном и том же триггерном событии. Если это событие происходит, и оба сторожевые условия имеют значение "истина", то между этими переходами имеет место конфликт, поскольку сработать может только один переход. Другими словами, в случае конфликтующих переходов только один из них может сработать на одном шаге выполнения до завершения. В рассматриваемом примере наступление события, которым является поступление обычного запроса на формирование отчета, приводит к конфликту переходов.

Для устранения конфликта переходов, если это возможно, следует видоизменить диаграмму конечного автомата. Так, например, измененный фрагмент диаграммы (см. рис. 10.10, *б*) не содержит конфликтующих переходов, поскольку сторожевые условия для переходов с одним именем триггерного события являются взаимно исключающими.

В ситуациях, когда существуют конфликтующие переходы, а диаграмму конечного автомата нельзя видоизменить без потери адекватности модели, выбор перехода, который должен сработать, может быть основан на неявном приоритете. Неявные приоритеты могут устранить некоторые конфликты переходов, но не все из них. Неявные приоритеты конфликтующих переходов основываются на относительной позиции этих переходов в иерархии состояний.

По определению, переход, выходящий из некоторого подсостояния, имеет более высокий приоритет, чем любой другой конфликтующий с ним переход, выходящий из любых содержащих его состояний. Тем самым приоритет перехода определяется на основе его состояния-источника. Приоритет соединенных переходов основывается на приоритете перехода с транзитивно наиболее вложенным состоянием-источником.

В общем случае, если `t1` является переходом, источником которого выступает состояние `s1`, и `t2` имеет источником `s2`, то:

- если `s1` является прямым или транзитивно вложенным подсостоянием `s2`, то `t1` имеет более высокий приоритет, чем `t2`;
- если `s1` и `s2` не входят в одну и ту же конфигурацию состояний, то не существует различия между приоритетами `t1` и `t2`.

Для выбора перехода из множества разрешенных переходов, который должен сработать в некоторой активной конфигурации, предназначен следующий алгоритм, основанный на неявных приоритетах. Множество переходов, которые могут сработать, является некоторым максимальным множеством переходов, которые удовлетворяют следующим условиям:

- все переходы в этом множестве являются разрешенными;
- не существует конфликтующих переходов в этом множестве;
- не существует перехода за пределами этого множества, который имеет более высокий приоритет, чем некоторый переход в этом множестве. Другими словами, разрешенные переходы с наивысшими приоритетами входят в это множество, в то время как конфликтующие переходы с более низкими приоритетами не входят в него.

Согласно этому алгоритму, состояния в активной конфигурации целесообразно рассматривать, начиная с самых внутренних вложенных простых состояний и продолжая их рассмотрение по направлению из вне. Для каждого

состояния на данном уровне оцениваются все выходящие переходы, чтобы определить, является ли они разрешенными. Такая последовательность рассмотрения гарантирует, что принцип приоритета не будет нарушен.

Наличие на диаграмме конечного автомата конфликтующих переходов с одинаковыми неявными приоритетами может привести к ее несогласованности. Единственный нетривиальный вопрос заключается в разрешении конфликтов переходов, пересекающих ортогональные состояния по всем уровнях. Он может быть решен введением правила окончания рассмотрения в каждом ортогональном состоянии сразу после того, как сработает переход внутри любого одного из его ортогональных регионов.

10.5. Композитные состояния и регионы

При рассмотрении состояний и переходов в этой главе неоднократно встречались понятия композитного состояния и региона. В данном разделе приводится подробное описание этих понятий и их свойств.

10.5.1. Основные определения

Композитное состояние (composite state) — состояние, содержащее в своем составе один регион или несколько ортогональных регионов.

Регион (region) — специальный элемент модели, который содержит состояния и переходы и является частью композитного состояния или конечного автомата.

Композитным состоянием является или некоторое простое композитное состояние с всего лишь одним регионом, или некоторое ортогональное композитное состояние с более чем одним регионом. Каждый регион имеет множество взаимно непересекающихся вершин и множество переходов.

Если регион является собственностью конечного автомата, то он не может одновременно находиться в собственности композитного состояния, и наоборот. Обычное или простое композитное состояние может содержать не более одного региона и используется только для моделирования последовательного поведения. Для моделирования параллельного поведения используются ортогональные композитные состояния и регионы.

Ортогональное (orthogonal) композитное состояние — композитное состояние, содержащее более одного региона, которые в этом случае называются *ортогональными регионами (orthogonal regions)*.

Таким образом, любое композитное состояние либо содержит один регион, либо декомпозируется на два или более ортогональных региона.

Любое состояние, заключенное в регион композитного состояния, называется *подсостоянием (substate)* этого композитного состояния. Оно называется *прямым подсостоянием (direct substate)*, если оно не содержится в никаком другом состоянии; в противном случае оно называется *непрямым подсостоянием (indirect substate)*.

Композитное состояние или конечный автомат с одним регионом изображается вложением в соответствующий графический символ диаграммы конечного автомата для этого региона (рис. 10.11, *а*). Имя композитного состояния может быть записано либо в верхней вкладке, либо внутри символа состояния. Композитное состояние может иметь одну или несколько точек входа и выхода на своей границе или внутри этой границы.

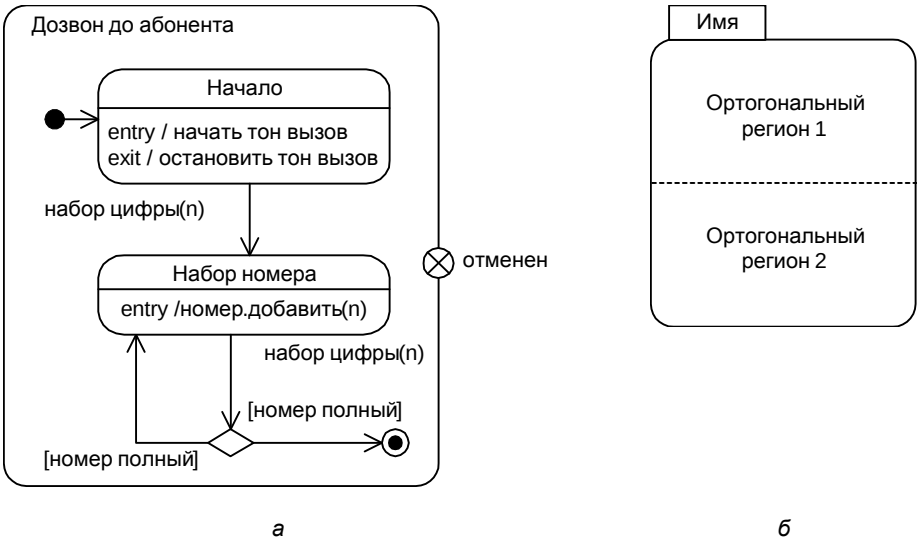


Рис. 10.11. Пример композитного состояния с одним регионом (*а*) и нотация ортогональных регионов (*б*)

Композитное состояние или конечный автомат с ортогональными регионами изображается разделением графического символа состояния или конечного

автомата пунктирными линиями на регионы (рис. 10.11, *б*). Имя ортогонального композитного состояния может быть записано только в верхней вкладке. Каждый регион может иметь необязательное имя и содержать вложенные состояния и переходы между ними.

Семантика регионов тесно связана с состояниями или конечными автоматами, имеющими регионы, и поэтому она определяется как часть семантики для состояния и конечного автомата. Следствием этого являются следующие ограничения. Регион может иметь не более одной начальной вершины, не более одной вершины неглубокой или глубокой истории.

Если конечный автомат является иерархическим, т. е. содержит некоторую иерархию вложенных композитных состояний, то в один и тот же момент времени может быть активным более чем одно состояние. Если при этом конечный автомат находится в простом состоянии, которое содержится в некотором композитном состоянии, то все композитные состояния, которые непосредственно или транзитивно содержат это простое состояние, также являются активными.

Кроме этого, поскольку конечный автомат в целом и некоторые из композитных состояний в этой иерархии могут быть ортогональными, т. е. содержать несколько регионов, текущее активное "состояние" в действительности представляется множеством деревьев состояний, начиная с наиболее верхних состояний регионов его корня и продолжая вниз до самого внутреннего активного подсостояния. Такое дерево состояний называется *конфигурацией состояний*.

Для конфигурации состояний имеют место следующие инварианты:

- если композитное состояние является активным и не ортогональным, то активным является не более одного из его подсостояний;
- если композитное состояние является активным и ортогональным, то все его регионы являются активными, при этом активным является не более чем одно подсостояние в каждом регионе.

10.5.2. Вход и выход в простом композитном состоянии

При рассмотрении входа в простое композитное состояние применяется *правило входа по умолчанию*, которое формулируется в следующей форме: если регион простого композитного состояния имеет начальное псевдосостояние, то любой внешний переход в это композитное состояние представляет переход в начальное псевдосостояние его региона.

Если регион простого композитного состояния не имеет начального псевдосостояния, то интерпретация любого внешнего перехода в это композитное состояние является точкой изменения семантики.

В отсутствие дополнительной спецификации рассмотренная ситуация может привести к несогласованной (ill-formed) модели. Таким образом, для региона наличие начального или исторического псевдосостояния является обязательным.

При входе в простое композитное состояние необходимо различать следующие случаи.

- Вход по умолчанию. Графически это изображается некоторым входящим переходом, который оканчивается на границе символа данного композитного состояния. В этом случае применяется правило входа по умолчанию. Если существует некоторое сторожевое условие на триггере этого перехода, оно должно иметь значение "истина".
- Явный вход. Если переход входит в некоторое подсостояние простого композитного состояния, то такое подсостояние становится активным, а его входное действие выполняется после выполнения входного действия композитного состояния. Это правило применяется рекурсивно, если переход оканчивается на транзитивно вложенном подсостоянии.
- Вход в неглубокую историю. Если переход оканчивается на псевдосостоянии неглубокой истории, то активным подсостоянием становится наиболее последнее активное подсостояние до этого входа, за исключением, когда наиболее последним активным подсостоянием является финальное состояние или если это есть первый вход в данное композитное состояние. В последних двух случаях вводится в рассмотрение *историческое состояние по умолчанию*, как подсостояние, которое является целью неявного перехода из этого псевдосостояния истории.
- Вход в глубокую историю. Правило здесь такое же, что и для неглубокой истории, за исключением того, что это правило применяется рекурсивно ко всем уровням в этой конфигурации активных состояний, начиная от уровня рассматриваемого псевдосостояния глубокой истории и ниже.
- Вход в точку входа. Если переход входит в простое композитное состояние через точку входа, то выполняется входное поведение композитного состояния до действия, ассоциированного с тем внутренним переходом, который выходит из этой точки входа.

При рассмотрении выхода из простого композитного состояния применяется *правило выхода по умолчанию*, которое формулируется в следующей форме: если регион простого композитного состояния имеет финальное состояние,

то переход в финальное состояние означает завершение поведения в регионе простого композитного состояния и инициирует событие завершения.

При выходе из простого композитного состояния необходимо различать следующие случаи.

- Выход по умолчанию. Графически это изображается некоторым выходящим переходом, который берет начало на границе символа простого композитного состояния и не содержит имени триггерного события. В этом случае применяется правило выхода по умолчанию.
- Явный выход. Графически это также изображается выходящим переходом, который берет начало на границе символа простого композитного состояния и содержит имя некоторого триггерного события. Такой переход также называют *внешним* (external) переходом. При этом выполняются выходные действия композитного состояния после выполнения выходных действий подсостояния источника. Это правило применяется рекурсивно, если переход выходит из транзитивно вложенного композитного состояния.
- Выход по верхнеуровневому переходу. Если переход выходит из некоторого подсостояния и пересекает границу простого композитного состояния, то срабатывание такого перехода приводит к выходу из этого композитного состояния и делает активным целевое состояние этого перехода. Такой переход также называется *верхнеуровневым* переходом. Когда происходит выход из простого композитного состояния по верхнеуровневому переходу, то рекурсивно происходит выход из любого его активного подсостояния. Это означает, что выходные действия выполняются в последовательности, начиная с самого внутреннего активного подсостояния в текущий конфигурации состояний. Если существует некоторое сторожевое условие на триггере этого перехода, оно должно иметь значение "истина".
- Выход из точки выхода. Если в простом композитном состоянии происходит выход через точку выхода, то выходные действия композитного состояния выполняются перед действиями, ассоциированными с переходом, выходящим из этой точки выхода.

Верхнеуровневый переход с целью за пределами данного композитного состояния означает выполнение выходного действия этого композитного состояния, в то время как верхнеуровневый переход с целью внутри данного композитного состояния не означает выполнения выходного действия этого композитного состояния.

10.5.3. Вход и выход в ортогональном композитном состоянии

Вход в ортогональное композитное состояние и выход из него осуществляются в соответствии с рассмотренными ранее правилами входа и выхода простого композитного состояния, с учетом наличия в нем нескольких ортогональных регионов.

Так, например, если каждый регион ортогонального композитного состояния имеет начальное псевдосостояние, то используется *правило входа по умолчанию для ортогональных регионов*, которое формулируется в следующей форме: если каждый регион ортогонального композитного состояния имеет начальное псевдосостояние, то любой внешний переход в это композитное состояние представляет переход в каждое из этих начальных псевдосостояний регионов.

Аналогично, при рассмотрении выхода из ортогонального композитного состояния применяется *правило выхода по умолчанию для ортогональных регионов*, которое формулируется в следующей форме: если каждый регион ортогонального композитного состояния имеет финальное состояние, то переход в каждое из финальных состояний означает завершение поведения в каждом регионе ортогонального композитного состояния и инициирует событие завершения.

Переход в финальное состояние представляет завершение поведения в охватываемом регионе. Завершение поведения во всех ортогональных регионах представляет завершение поведения в охватываемом состоянии и инициирует событие завершения в охватываемом состоянии.

Всякий раз, когда происходит вход в некоторое ортогональное композитное состояние, также выполняется вход в каждый из его ортогональных регионов или по умолчанию, или явно. Если переход оканчивается на границе этого композитного состояния, то происходят входы во все регионы с использованием входов по умолчанию. Если переход явно входит в один или более регионов с использованием узла разделения, то происходят входы в эти регионы: в одни — явно, а другие — по умолчанию.

Когда происходит выход из ортогонального композитного состояния, осуществляются выходы из всех его ортогональных регионов. После этого выполняется выходная деятельность этого композитного состояния.

В качестве примера композитного состояния с ортогональными регионами рассмотрим диаграмму конечного автомата, которая описывает прохождение некоторого курса студентами (рис. 10.12). Эта диаграмма изображена во фрейме и содержит композитное состояние с именем `Обучение`. Последнее,

в свою очередь, содержит три ортогональных региона, в каждый из которых выполняется вход по умолчанию. Выход из этого композитного состояния может выполняться по умолчанию или явно. В первом случае необходимо завершение всех ортогональных регионов, что соответствует успешному прохождению курса. Во втором случае выход инициализируется только из одного региона после неудачной сдачи экзамена, что соответствует неудовлетворительному прохождению курса.

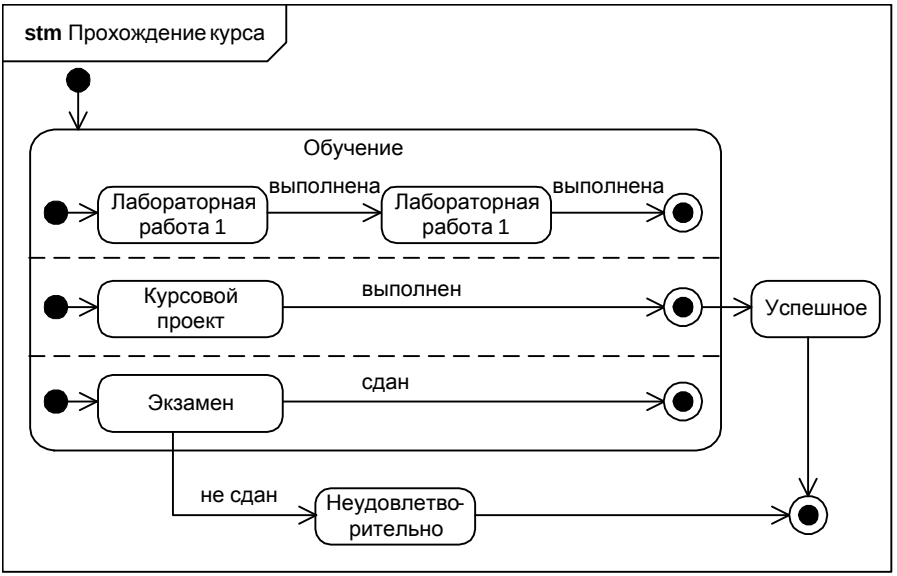


Рис. 10.12. Пример композитного состояния с ортогональными регионами

Рассмотренный пример диаграммы конечного автомата не учитывает многих особенностей реального прохождения курса на практике, таких как, например, пересдача экзамена или необходимость сдачи лабораторных работ и защиты курсового проекта перед сдачей экзамена. Читателям предлагается самостоятельно модифицировать данную диаграмму с целью учета этих особенностей в качестве упражнения.

10.5.4. Скрытая секция декомпозиции

В дополнение к секциям, которые были рассмотрены ранее для простого состояния, композитное состояние может явно иметь секцию декомпозиции. Эта секция содержит вложенную диаграмму, которая показывает структуру внутреннего поведения в терминах регионов, состояний и перехода (см. рис. 10.12).

Иногда удобно скрыть декомпозицию композитного состояния. Например, композитное состояние может содержать большое число подсостояний, которые не могут быть размещены в графической области диаграммы без потери наглядности. В этом случае композитное состояние может быть представлено графическим символом простого состояния с дополнительной пиктограммой скрытой декомпозиции, которая обычно размещается в правом нижнем углу (рис. 10.13, *а*).

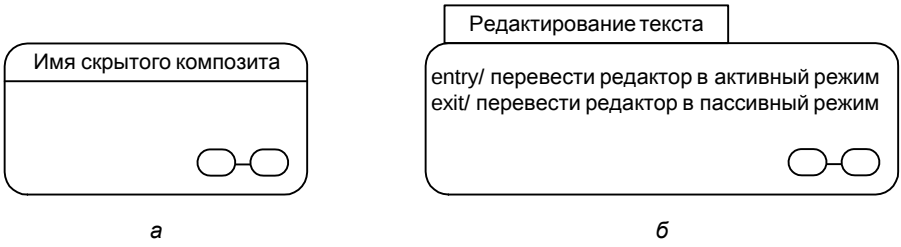


Рис. 10.13. Композитные состояния с пиктограммой скрытой декомпозиции: *а* — общий вид; *б* — пример использования

Эта пиктограмма, состоящая из двух горизонтально размещенных и соединенных состояний, является необязательным визуальным индикатором того, что состояние имеет некоторую декомпозицию, которая не изображена на данной диаграмме конечного автомата. Вместо этого содержимое данного композитного состояния может быть изображено на отдельной диаграмме.

Так, например, композитное состояние `Редактирование текста` из рассмотренной ранее диаграммы конечного автомата (см. рис. 10.8) может быть представлено в форме скрытой декомпозиции так, как показано на рис. 10.13, *б*.

В каждом ортогональном регионе с активной конфигурацией состояний, который не декомпозируется на ортогональные регионы, т. е. является регионом нижнего уровня, в результате текущего наступления события может сработать не более одного перехода. Когда все ортогональные регионы закончат выполнение перехода, текущее наступление события полностью поглощается, и шаг выполнения до завершения заканчивается.

Конечный автомат допускает возможность обобщения. Специализированный конечный автомат является расширением общего конечного автомата, он должен иметь все элементы общего конечного автомата и может иметь дополнительные элементы, такие как регионы, вершины и переходы.

Отдельные состояния конечного автомата могут быть переопределены посредством добавления состояний и переходов. Могут быть переопределены и переходы. Простое состояние может быть переопределено или расширено

посредством композитного состояния, а также посредством добавления одного или более регионов. Композитное состояние может быть переопределено или расширено либо посредством наследуемых регионов, либо посредством добавления регионов. Регион расширяется с помощью добавления вершин, состояний и переходов, а также через переопределения состояний и переходов.

10.6. Состояние подавтомата

Состояние подавтомата (submachine state) семантически эквивалентно композитному состоянию, определенному в некотором конечном автомате, на который ссылается это состояние.

Вход в состояние подавтомата и выход из него осуществляется, в противоположность обычному композитному состоянию, посредством точек входа и выхода. По умолчанию вход в некоторое состояние подавтомата может произойти посредством его начального псевдосостояния или посредством любой из его точек входа. При этом может произойти вход в некоторое неортогональное или ортогональное композитное состояние. Вход посредством начального псевдосостояния имеет тот же смысл, что и для обычных композитных состояний.

Состояние подавтомата изображается с помощью символа простого состояния, строка в секции имени которого имеет следующий синтаксис:

`<имя-состояния>':'<имя-конечного-автомата>`

где `<имя-конечного-автомата>` соответствует конечному автомату, на который ссылается это состояние подавтомата.

Символ состояния подавтомата может содержать одну или несколько точек входа и одну или несколько точек выхода, которые изображаются на границе состояния подавтомата. Именами этих точек являются имена соответствующих точек входа или выхода, которые определены в ссылаемом конечном автомате.

Если вход в конечный подавтомат происходит по умолчанию через его начальное псевдосостояние, или если выход из него происходит в результате завершения этого подавтомата, использовать нотацию точек входа/выхода нет необходимости. Аналогично, точка выхода не требуется, если выход имеет место через явный переход, который берет начало на границе этого состояния подавтомата. В последнем случае предполагается, что он применяется ко всем подсостояниям этого состояния подавтомата.

В качестве примера рассмотрим состояние подавтомата, которое ссылается на некоторый конечный автомат отказов (рис. 10.14). Соответствующий

конечный автомат может быть определен на отдельной диаграмме конечного автомата. В этом примере переход, срабатывающий при событии `ошибка 1`, влечет срабатывание перехода по умолчанию в конечном автомате отказов. Переход, срабатывающий при событии `ошибка 2`, будет завершен в узле входа конечного автомата отказов. Переход, выходящий из узла выхода этого состояния подавтомата, должен выполнить действие `обработка неполная` в дополнение к действиям, которые выполняются в этом состоянии подавтомата. Наконец, переход, выходящий из границы состояния подавтомата, выполняется в результате события завершения, сгенерированного, когда состояние подавтомата достигнет своего финального состояния.

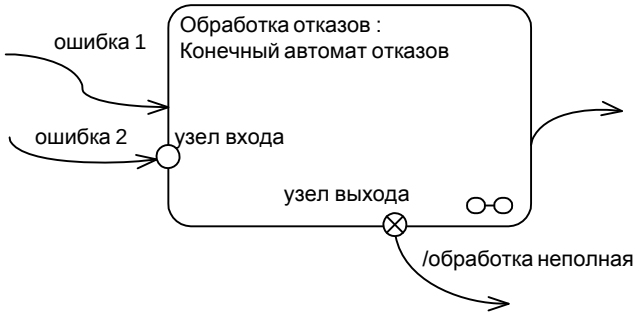


Рис. 10.14. Пример изображения состояния подавтомата

Следует заметить, что такая же нотация могла бы быть применена для композитных состояний с исключением, которое не имеет ссылки на конечный автомат в своем имени состояния.

Состояние подавтомата специфицирует объявление спецификации состояния подавтомата в конечном автомате. Конечный автомат, который содержит состояние подавтомата, называется *контейнерным* конечным автоматом. Один и тот же конечный автомат может быть подавтоматом более чем один раз в контексте одного контейнерного конечного автомата.

Регионы состояния подавтомата в конечном автомате являются регионами этого композитного состояния. Действия входа, выхода и поведения, а также внутренние переходы определяются как часть этого состояния. Состояние подавтомата является механизмом декомпозиции, который позволяет разложить общее поведение и повторно его использовать. Переходы в контейнерном конечном автомате могут иметь точки входа и выхода во вложенный конечный автомат в качестве целей и источников соответственно.

В случае, когда композитное состояние является ортогональным, точка входа эквивалентна псевдосостоянию разделения.

Как и для начальных переходов по умолчанию, сторожевые условия, ассоциированные с триггерами переходов, выходящих из точки входа, должны принимать значение "истина". В противном случае модель является несогласованной (ill-formed).

Аналогично, выход из состояния подавтомата с ортогональными регионами может быть результатом достижения финального состояния в каждом из ортогональных регионов посредством некоторого "группового" перехода или посредством любой из точек выхода. Выход с помощью финального состояния (группового перехода) имеет тот же смысл, что и для обычных композитных состояний. В случае, когда состояние подавтомата является ортогональным, точка выхода эквивалентна псевдосостоянию слияния. Выход посредством такой точки выхода означает, что первым выполняется действие перехода с точкой выхода в качестве цели, после чего выполняется выходное действие данного состояния подавтомата.

В качестве примера использования состояния подавтомата рассмотрим диаграмму конечного автомата для банкомата (рис. 10.15). На этой диаграмме состояние подавтомата с именем `Ввод суммы` ссылается на конечный автомат с именем `Ввод суммы` `ATM` (см. рис. 10.6, *в*). Заметим, что точка входа здесь не применяется, что не исключает возможность ее использования на других диаграммах в соответствующем контексте.

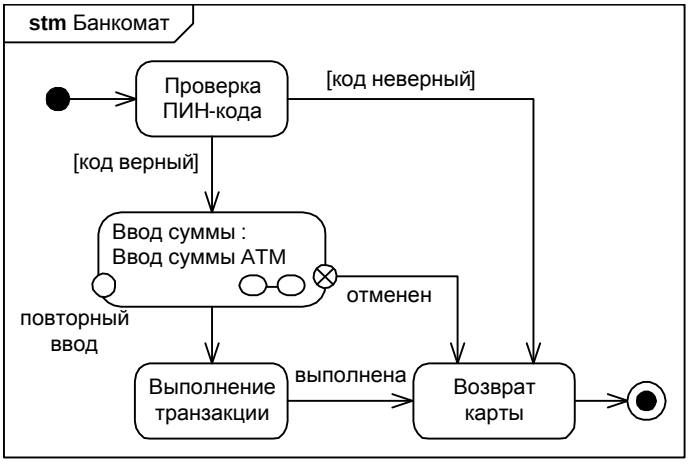


Рис. 10.15. Состояние подавтомата с использованием точки выхода

Состояние подавтомата может быть переопределено. В этом случае состояние подавтомата может быть заменено другим состоянием подавтомата

в предположении, что оно имеет такие же точки входа и выхода, как и переопределяемое состояние подавтомата. При этом оно может иметь и дополнительные точки входа и выхода. Для переходов могут быть переопределены действия на переходе и целевое состояние, в то время как состояние-источник и триггер остаются без изменения.

10.7. Протокольный конечный автомат

Концепция конечного автомата является удобным формализмом для спецификации не только жизненного цикла объектов, но и особенностей вызова его операций.

Протокольный конечный автомат (protocol state machine) предназначен для спецификации допустимых состояний и условий при вызове операций классификатора.

Протокольный конечный автомат всегда определяется в контексте некоторого классификатора. Он представляет возможные и разрешенные переходы для экземпляров классификатора его контекста совместно с операциями, которые могут представляться переходами. Таким образом, жизненный цикл некоторого экземпляра может быть описан в форме протокольного конечного автомата для его классификатора посредством спецификации порядка, в котором могут быть вызваны операции, и состояний, которые представляют изменения отдельных характеристик этого экземпляра.

Графическая нотация для протокольного конечного автомата практически совпадает с нотацией конечных автоматов поведения. Дополнением является ключевое слово `{protocol}`, которое помещается рядом с именем этого конечного автомата. Это дополнение позволяет визуально отличать диаграммы протокольных конечных автоматов от диаграмм конечных автоматов поведения (рис. 10.16).

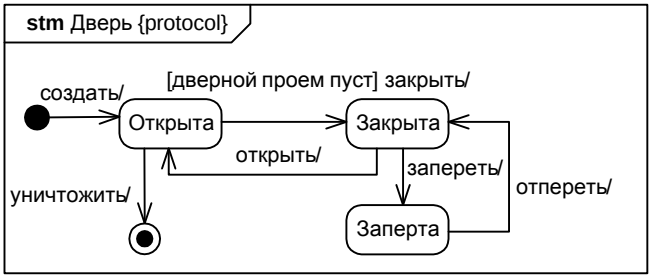


Рис. 10.16. Пример протокольного конечного автомата

Протокольные конечные автоматы позволяют определить способы использования операций классификатора и его клиентов посредством спецификации:

- контекста, а именно, в каких состояниях и при каких предусловиях могут быть вызваны операции объекта;
- протокольного порядка между вызовами операций объекта;
- ожидаемого результата использования операций.

В протокольном конечном автомате также изображаются ограничения, которые могут быть специфицированы для предусловий и постусловий операций. Эти условия следует принимать во внимание при вызове операций объекта. Протокольный конечный автомат содержит специальные протокольные состояния и протокольные переходы.

Для протокольных конечных автоматов могут быть также использованы составные переходы, конечные подавтоматы, композитные состояния и параллельные регионы. Классификатор может иметь несколько протокольных конечных автоматов.

10.7.1. Протокольное состояние

Протокольное состояние (protocol state) — состояние протокольного конечного автомата, которое представляет некоторую открытую для наблюдения стабильную ситуацию контекста его классификатора.

Протокольные состояния обеспечивают некоторое внешнее представление класса, который открывается для наблюдения со стороны своих клиентов. Если экземпляр классификатора не выполняет никакой операции, то клиенты этого экземпляра могут всегда знать его текущее состояние. Для состояний протокольного конечного автомата существуют два отличия по сравнению с состояниями конечного автомата поведения:

- для протокольных конечных автоматов не существуют отдельные характеристики конечного автомата поведения (например, `entry`, `exit`, `do`);
- состояния в протокольных конечных автоматах могут иметь некоторый инвариант. Выражение этого инварианта может быть представлено в форме строки текста, размещенной после или ниже имени состояния и заключенной в квадратные скобки.

Для протокольных конечных автоматов имеют место следующие ограничения:

- протокольный конечный автомат может иметь только контекст классификатора, но не контекст характеристики поведения;

- все переходы протокольного конечного автомата должны быть протокольными переходами;
- состояния протокольного конечного автомата не могут иметь действий входа, выхода или "ду-деятельность";
- протокольные конечные автоматы не могут иметь псевдосостояния глубокой или неглубокой истории.

Протокольный конечный автомат определяет допустимые переходы при выполнении операций объекта. В этом случае протокольный конечный автомат должен представлять все операции, которые могут служить причиной изменения состояний для объектов класса. Те операции, которые не генерируют никаких переходов, в протокольном конечном автомате не изображаются.

10.7.2. Протокольный переход

Протокольный переход (protocol transition) специфицирует некоторый законный переход для операции, представленной в форме протокольного конечного автомата.

Переходы протокольных конечных автоматов имеют следующую информацию: предусловие в форме сторожевого условия, триггер и постусловие. Каждый протокольный переход ассоциируется с ноль или одной операцией, которая принадлежит контексту классификатора протокольного конечного автомата. В этом случае протокольный переход специфицирует тот факт, что соответствующая операция может быть вызвана для экземпляра в исходном состоянии при начальном условии в форме сторожевого условия, и что в конце этого перехода будет достигнуто состояние назначения при финальном условии в форме постусловия.

Для графического представления протокольного перехода применяется обычная нотация перехода конечного автомата поведения. Отличие заключается в том, что для протокольных переходов не специфицируются никакие действия, и для них могут существовать постусловия. Постусловия имеют тот же синтаксис, что и сторожевые условия, но представляются в конце синтаксиса перехода (рис. 10.17, *а*).

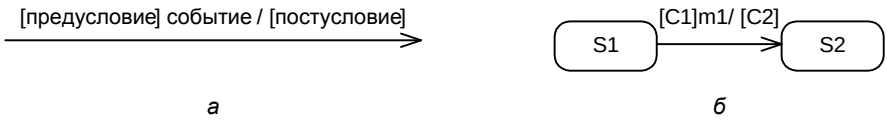


Рис. 10.17. Нотация протокольного перехода (*а*) и пример протокольного перехода, ассоциированного с операцией `m1` (*б*)

Предусловие специфицирует условие данного перехода, которое должно быть проверено до срабатывания перехода. Это сторожевое условие, добавленное к состоянию-источнику, должно оцениваться как часть предусловия операции, на которую ссылается переход. Постусловие специфицирует условие данного перехода, которое должно быть получено сразу же после срабатывания перехода. Это постусловие является частью постусловия операции, ассоциированной с этим переходом.

Интерпретация протокольных переходов выполняется, как правило, в терминах предусловий и постусловий соответствующей операции. Например, переход на рис. 10.17, *б* может быть интерпретирован следующим образом:

- операция `m1` может быть вызвана у экземпляра класса, когда он находится в протокольном состоянии `S1` при выполнении условия `C1`;
- при вызове операции `m1` в протокольном состоянии `S1` при условии `C1` должно быть достигнуто протокольное состояние `S2` при выполнении условия `C2`.

Для протокольных переходов имеют место следующие ограничения:

- протокольный переход может принадлежать только протокольному конечному автомату;
- протокольный переход никогда не имеет действий на переходе;
- если протокольный переход ссылается на операцию, т. е. имеет триггер вызова операции, соответствующий некоторой операции, то эта операция должна применяться в контексте классификатора конечного автомата, содержащего данный протокольный переход;
- если на некоторую операцию объекта не ссылается никакой протокольный переход, то такая операция может быть вызвана для любого состояния протокольного конечного автомата, но она не изменяет текущее состояние.

Хотя действие для протокольного перехода не специфицируется, тем не менее оно является неявным, если переход имеет триггер вызова операции. В этом случае действию на переходе соответствует операция, специфицированная этим триггером вызова операции.

Интерпретация события приема сигнала в некотором текущем состоянии, при инварианте состояния или при предусловии является точкой изменения семантики. Это событие может быть проигнорировано, отвергнуто или отложено, либо может порождаться некоторое исключение, или приложение может быть остановлено по причине ошибки. Семантически это соответствует нарушению некоторого предусловия, для которого никакого поведения в UML 2.0 не определено.

Протокольные конечные автоматы могут быть переопределены в более специальные протокольные конечные автоматы, или в конечные автоматы пове-

дения. Соответствие протоколу объявляет, что специальный протокольный конечный автомат специфицирует некоторый протокол, который согласуется с протоколом общего конечного автомата, или что специальный конечный автомат поведения действует в соответствии с протоколом общего протокольного конечного автомата.

Поскольку протокольный конечный автомат является собственностью некоторого классификатора, то классификаторы, владеющие некоторым общим конечным автоматом и некоторым ассоциированным специальным конечным автоматом, находятся в отношении обобщения или реализации.

* * *

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм конечного автомата (табл. 10.1).

Таблица 10.1. *Графические узлы и пути на диаграммах конечного автомата*

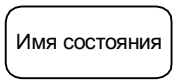
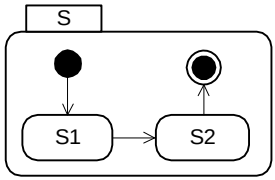
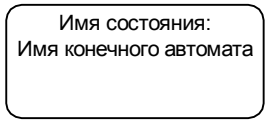
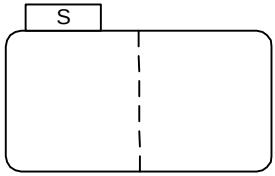
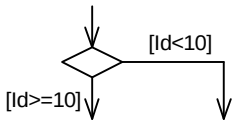
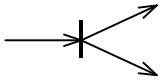
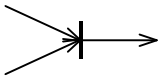
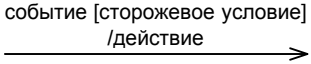
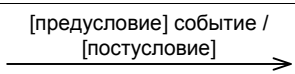
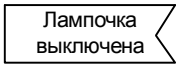
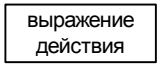
Тип графического элемента	**Нотация**	**Ссылка на раздел**
Простое состояние (simple state)		Простое состояние *(см. разд. 10.2)*
Композитное состояние (composite state)		Композитные состояния и регионы *(см. разд. 10.5)*
Состояние подавтомата (submachine state)		Состояние подавтомата *(см. разд. 10.6)*
Регион (region)		Композитные состояния и регионы *(см. разд. 10.5)*
Начальное псевдосостояние (initial pseudo state)		Начальное псевдосостояние *(см. разд. 10.3.1)*

Таблица 10.1 *(окончание)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Псевдосостояние соединения (junction pseudo state)		Соединение *(см. разд. 10.3.4)*
Узел завершения (terminate node)		Узел завершения *(см. разд. 10.3.2)*
Финальное состояние (final state)		Финальное состояние *(см. разд. 10.3.11)*
Псевдосостояние неглубокой истории (shallow pseudo state)		Неглубокая история *(см. разд. 10.3.9)*
Псевдосостояние глубокой истории (deep pseudo state)		Глубокая история *(см. разд. 10.3.10)*
Точка входа (entry point)		Точка входа *(см. разд. 10.3.7)*
Точка выхода (exit point)		Точка выхода *(см. разд. 10.3.8)*
Псевдосостояние выбора (choice pseudo state)		Выбор *(см. разд. 10.3.3)*
Вершина разделения (fork vertex)		Разделение *(см. разд. 10.3.5)*
Вершина слияния (join vertex)		Слияние *(см. разд. 10.3.6)*
Переход (transition) конечного автомата поведения		Переход *(см. разд. 10.4)*
Протокольный переход (protocol transition)		Протокольный переход *(см. разд. 10.7.2)*
Действие передачи сигнала (send signal action)		Передача сигнала *(см. разд. 10.4.4)*
Действие приема сигнала (receive signal action)		Прием сигнала *(см. разд. 10.4.5)*
Действие (action)		Действия на переходе *(см. разд. 10.4.6)*

Глава 11

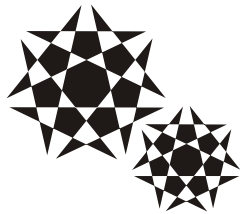


Диаграмма компонентов (component diagram)

Все рассмотренные ранее диаграммы отражали концептуальные аспекты моделирования структуры и поведения системы и относились к логическому уровню представления. Особенность логического представления заключается в том, что оно оперирует понятиями, которые имеют в некотором смысле виртуальный характер. Другими словами, различные элементы логического представления, такие как классы, ассоциации, состояния и сообщения, не являются материальными сущностями. Они лишь отражают понимание структуры или архитектуры той или иной системы или динамические аспекты ее поведения.

Основное назначение логического представления состоит в анализе структурных и функциональных отношений между элементами модели системы. Однако для создания конкретной физической системы необходимо некоторым образом реализовать все элементы логического представления в форме конкретных материальных сущностей. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно — физический уровень представления.

11.1. Особенности физического моделирования в языке UML 2.0

Физическое моделирование в языке UML 2.0 предназначено для спецификации программных компонентов и взаимосвязей между ними, а также артефактов, которые их реализуют, и узлов, на которых они развертываются и выполняются. Одним из основных понятий физического моделирования является компонент, под которым понимается модульная единица программы с возможными интерфейсами. Для графического представления компонентов

и взаимосвязей между ними предназначена диаграмма компонентов (Component diagram), которая рассматривается в данной главе.

Другим основным понятием физического моделирования является узел, на котором развертывается один или несколько компонентов. Для графического представления узлов и взаимосвязей между ними предназначена диаграмма развертывания (deployment diagram), которая рассматривается в *главе 12*.

Чтобы пояснить отличие физического от логического представления, рассмотрим в общих чертах процесс разработки некоторой программной системы. Ее исходным логическим представлением могут служить диаграммы структуры классов и пакетов, диаграммы поведения и взаимодействий, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на некотором языке программирования (C++, C#, Delphi/Pascal, Basic/VBA, Java). При этом уже в тексте программы предполагается некоторая организация программного кода, определяемая синтаксисом языка программирования и предполагающая разбиение исходного кода на отдельные модули.

Тем не менее исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат важным аспектом его физического представления. Очевидно, программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов и процедур, форм графических интерфейсов, файлов таблиц баз данных. Именно эти компоненты являются базовыми элементами физического представления системы в нотации языка UML.

Диаграмма компонентов разрабатывается для следующих целей:

- ❒ визуализация общей структуры исходного кода программной системы;
- ❒ спецификация исполнимого варианта программной системы;
- ❒ обеспечение многократного использования отдельных фрагментов программного кода;
- ❒ представление концептуальной и физической схем баз данных.

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие — на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве классификаторов.

Спецификация компонентов в модели программной системы позволяет их повторно использовать в различных проектах, заменяя и связывая их вместе посредством предоставляемых и требуемых интерфейсов. Эти аспекты автономности и повторного использования компонентов позволяют добиться гибкости и сокращения сроков разработки при выполнении реальных проектов. Артефакты, которые реализуют компоненты, предназначены для начального и повторного развертывания при их реализации с использованием наиболее современных программных технологий независимо, например, от модификации существующей системы.

Физическое моделирование концентрирует внимание не только на компонентах и связях между ними, но также и на особенностях программной реализации компонентов и их выполнения в системе. Понятие компонента как программного модуля позволяет определить его внешнюю спецификацию в форме одного или более предоставляемых и требуемых интерфейсов. При этом внутренняя архитектура компонентов инкапсулирована и может состоять из одного или более классов, которые реализуют его поведение.

На практике часто используется расширенное определение компонента как связанной группы элементов, что позволяет рассматривать его в качестве отдельной части или подсистемы разрабатываемого проекта. Понятие компонента как некоторой подсистемы наделяет его способностью иметь в собственности и импортировать потенциально открытое множество различных элементов модели. При этом отдельные части большой программной системы могут быть соединены вместе в форме повторно используемых компонентов, соединенных между собой посредством требуемых и предоставляемых интерфейсов. В языке UML 2.0 также определены специальные соединители для взаимосвязи компонентов на основе совместимости их интерфейсов.

В общем случае компонент может представлять собой сложный, физический, иногда распределенный архитектурный объект, который взаимодействует со своим окружением при помощи интерфейсов, которые соединяются с компонентом при помощи портов. Принято говорить, что компонент владеет всеми портами, которые содержатся или находятся в собственности компонента и не могут существовать независимо от него. Порты являются важными элементами в контексте моделирования распределенных систем и систем реального времени.

Хотя порт входит в состав компонента, он также является самостоятельным физическим объектом, для связи с которым должен быть реализован специальный интерфейс. Будучи физическими объектами, порты являются видимыми изнутри и извне компонентов. Для доступа к функциональности компонента необходимо указать роли, которые могут играть порты в окружении

компонента. Для этой цели предназначены спецификации портов и специальные протоколы.

Отношения на диаграмме компонентов моделируются при помощи соединителей и зависимостей. Соединитель представляет собой абстракцию физического канала связи, предоставляющего средства передачи для определенного абстрактного сигнального протокола. Соединители на диаграмме компонентов моделируются при помощи ассоциаций или связей. В этом случае компоненты связываются друг с другом исключительно посредством своих портов. Это позволяет отделить внутреннее представление компонентов от информации об их окружении, что делает возможным их многократное использование.

Для более детальной спецификации компонентов на практике могут быть использованы дополнительные стереотипы. Наиболее часто для компонентов используются следующие стереотипы: `<<subsystem>>` — для компонентов большого размера, `<<specification>>` — для спецификации компонента, `<<realization>>` — для реализации компонента. Последние два стереотипа применяются, когда одна спецификация может иметь несколько реализаций. Как правило, семантика стереотипов компонентов описывается в дополнительных спецификациях, которые получили название *профилей* (profiles).

Следует отметить, что в ближайшее время в рамках консорциума OMG (**www.omg.org**) предполагается разработать целый ряд новых профилей для специальных компонентных технологий и соответствующих платформ. Эти профили будут более детально учитывать особенности реализации компонентов на той или иной платформе или языке программирования.

Таким образом, полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. При этом диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых могут выступать различные артефакты. На практике компонент чаще всего соответствует отдельному файлу с исходным или исполняемым кодом. В этом случае отношения зависимости между компонентами часто интерпретируются аналогично тем, которые имеют место при компиляции исходных текстов программ.

Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы, порты и отношения между ними. Логично начать рассмотрение этих элементов с базового понятия, которым является компонент.

11.2. Компонент

Компонент (component) — элемент модели, представляющий некоторую модульную часть системы с инкапсулированным содержимым, спецификация которого является взаимозаменяемой в его окружении.

В языке UML 2.0 в качестве компонента может рассматриваться любой автономный элемент системы или подсистемы. Исторически понятие компонента относится к области компонентно-ориентированной разработки (Component-based Development, CBD) и структуризации системы. В этом контексте компонентная модель разрабатывается и используется на всем жизненном цикле системы и последовательно уточняется при ее развертывании и во время эксплуатации. Важным аспектом компонентной разработки является повторное использование ранее созданных компонентов.

Внутренние части компонентов скрыты и недоступны извне, кроме тех, которые предоставляются посредством его интерфейсов. Хотя компонент может зависеть от других элементов модели также в терминах интерфейсов, он является полностью инкапсулированным в объектно-ориентированном смысле. В языке UML 2.0 компоненты могут быть связаны между собой с помощью строго определенных отношений. Эти отношения определены таким образом, чтобы рассматривать компонент как можно более независимым от своего окружения.

В языке UML 2.0 компонент изображается в форме прямоугольника классификатора с ключевым словом `<<component>>`, записанным в форме стереотипа. Дополнительно в правом верхнем углу символа классификатора может быть изображена пиктограмма компонента в форме небольшого прямоугольника с двумя меньшими прямоугольниками, вставленными в его левую сторону (рис. 11.1, *а*). Внутри символа прямоугольника указывается имя компонента в соответствии с правилами записи имен классификатора. Хотя эта нотация в языке UML 2.0 определена как дополнительная, однако на практике она используется наиболее часто, если не сказать — повсеместно. Изображение компонента на диаграмме может содержать также символы интерфейсов, которые присоединяются к прямоугольнику компонента с помощью соединителей (рис. 11.1, *б*).

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и некоторых знаков препинания. Как будет отмечено далее, компонент может быть представлен на уровне типа или на уровне экземпляра. Хотя его графическое изо-

бражение в обоих случаях одинаковое, правила записи имени компонента несколько отличаются.

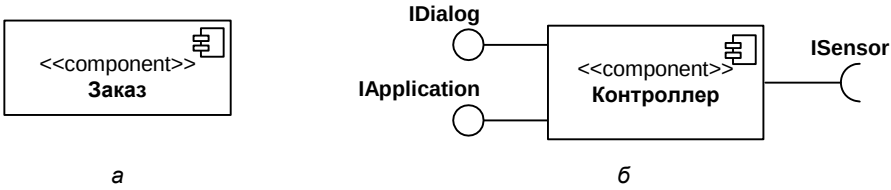


Рис. 11.1. Примеры изображения: *а* — простого компонента; *б* — компонента с интерфейсами

Внешнее представление компонента, посредством которого становятся доступными его видимые свойства и операции, получило специальное название — представление *черного ящика*. В этом представлении внутренняя структура компонента полностью скрыта от его окружения. Открытыми или доступными извне являются только интерфейсы компонента, которые могут быть указаны в форме свойств компонента. В этом случае может быть использована альтернативная нотация, при которой интерфейсы записываются в отдельной секции символа компонента (рис. 11.2, *а*).

Внутреннее представление компонента, визуализирующее его закрытые свойства и артефакты, получило специальное название — представление *белого ящика*. Это представление показывает, как специфицированное и видимое извне поведение компонента реализуется внутри него. В этом случае также может быть использована альтернативная нотация, при которой интерфейсы, закрытые свойства и артефакты записываются в отдельных секциях символа компонента (рис. 11.2, *б*).

Взаимосвязь между внешним и внутренним представлениями компонента устанавливается посредством отношений зависимости и специальных соединителей, которые рассматриваются далее в этой главе.

Вне зависимости от нотации, компонент может рассматриваться либо в качестве типа, либо в качестве экземпляра. Компонент в качестве *типа* наделяется возможностью иметь атрибуты и операции, а также участвовать в ассоциациях и обобщениях. Такой компонент может образовывать абстракцию для множества классификаторов, которые реализуют его поведение. При этом один компонент может быть заменен другим, только если они оба являются согласованными по типу.

Если компонент представляется на уровне типов, то в качестве его имени записывается только имя типа с заглавной буквы в форме <*имя-типа*>, аналогично

имени класса. В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации данного компонента. Необходимо заметить, что в этом случае номер версии записывается как помеченное значение в фигурных скобках.

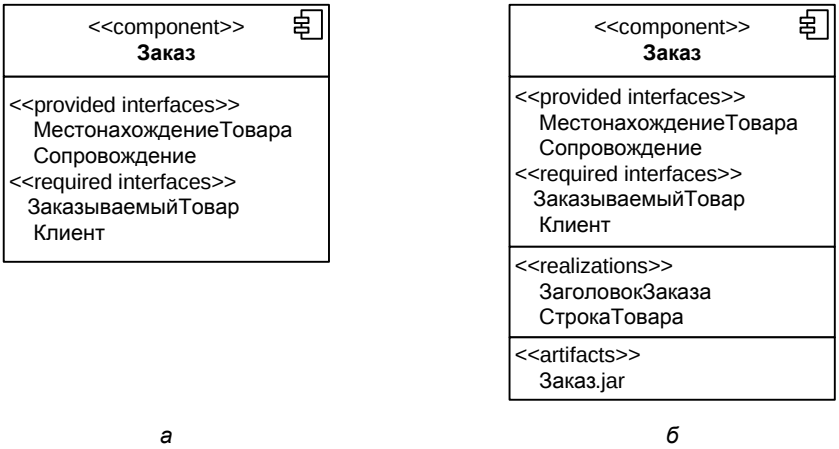


Рис. 11.2. Примеры изображения компонента в нотации:
а — черного ящика; *б* — белого ящика

Поскольку класс является инкапсулируемым классификатором, компонент может иметь необязательную внутреннюю структуру, а также некоторое множество интерфейсов и портов, которые специфицируют точки его взаимодействия. Компонент в качестве типа образует собственное пространство имен с возможностью импорта элементов. Тем самым в пространстве имен компонента все элементы, которые включены в его состав или относятся к его определению, должны находиться в его собственности или явно импортироваться.

На практике термин *"компонент"* используется в различном смысле. Одни разработчики применяют этот термин для обозначения любого программного модуля. Другие подразумевают под ним готовый объект, представляющий собой элемент развертывания. Третьи считают его файлом с исполнимым кодом, который может создавать экземпляры классов во время своего выполнения. Однако следует помнить, что в языке UML 2.0 определение компонента абстрагировано от любых особенностей его реализации.

В языке UML 2.0 компонент может включать в свой состав варианты использования, конечные автоматы, пакеты, а также другие компоненты и артефакты. В собственности компонента также может находиться некоторое множество реализаций. Реализации ссылаются на классификаторы, для которых данный

компонент является абстракцией и которые реализуют его поведение. Система может быть расширена посредством добавления новых типов компонентов, что добавляет новую функциональность.

Компонент в качестве *экземпляра* может быть инстанцирован от соответствующего компонента в качестве типа. На практике экземпляром компонента может быть некоторый объект времени выполнения или некоторый артефакт, который определяется только во время проектирования. Во втором случае объект, специфицированный компонентом, не существует во время выполнения программы. При этом неявно предполагается, что экземпляр компонента может быть получен посредством инстанцирования реализующих его классов или частей. Имя экземпляра компонента записывается аналогично имени линии жизни на диаграммах взаимодействия в следующем формате (БНФ):

```
<имя-экземпляра-компонента>::= [<собственное-имя-компонента>]
[':'<имя-типа>]
```

При этом собственное имя компонента записывается со строчной буквы, а в качестве имени экземпляра компонента должен присутствовать хотя бы один терм.

Поскольку физическое представление модели системы зависит от используемых аппаратно-программных технологий, имена компонентов могут зависеть от особенностей синтаксиса применяемых языков программирования и программных архитектур.

В качестве имен компонентов на практике часто используют имена исполняемых файлов, динамических библиотек, Web-страниц, текстовых файлов или файлов справки, имена файлов баз данных или файлов с исходными текстами программ. Примерами компонентов являются компоненты: COM+, .NET, VCL, CORBA, EJB, WSDL. Практически все коммерческие библиотеки также могут служить примерами компонентной разработки.

Символ компонента в нотации UML 2.0 отличается от обозначения компонента в языке UML 1.*x*, который больше не имеет двух выступающих прямоугольников. Возможно, они создавали неудобство для изображения символов интерфейсов и портов на границе компонента. Вместо этого в нотации UML 2.0 рекомендуется использовать ключевое слово <<component>> с дополнительной пиктограммой в правом верхнем углу. Для обратной совместимости с нотацией UML 1.*x* на диаграммах языка UML 2.0 допускается использовать старый символ с выступающими прямоугольниками.

Принято считать, что компонент специфицирует некоторый формальный контракт на сервисы, которые он предоставляет своим клиентам и которые он требует от других компонентов системы. Эти сервисы специфицируются в терминах предоставляемых и требуемых интерфейсов.

11.3. Интерфейс

Интерфейсы уже рассматривались ранее в *главах 4* и *5*, поэтому здесь будут отмечены те их особенности, которые характерны для их представления на диаграммах компонентов. Компонент может иметь две разновидности интерфейсов: предоставляемые и требуемые интерфейсы, которые образуют основу для взаимосвязей компонентов между собой с использованием зависимостей и соединителей.

Предоставляемый интерфейс (provided interface) — интерфейс, который компонент предлагает для своего окружения.

Требуемый интерфейс (required interface) — интерфейс, который необходим компоненту от своего окружения для выполнения заявленной функциональности, контракта или поведения.

Предоставляемые интерфейсы могут быть реализованы непосредственно компонентом или одним из классов, который находится в собственности этого компонента. Предоставляемые интерфейсы также могут предлагаться через общедоступные порты. Предоставляемые интерфейсы иногда называют *реализуемыми*, реже — *обеспеченными* интерфейсами.

Требуемые интерфейсы компонент запрашивает у других компонентов в окружающей его среде, поскольку компонент нуждается в них для обеспечения своих функциональных возможностей. Требуемые интерфейсы могут использоваться непосредственно компонентом или классами, которые находятся в собственности этого компонента. Они также могут быть интерфейсами, которые требуются его общедоступным портам.

Предоставляемые и требуемые интерфейсы в языке UML 2.0 изображаются с использованием специальной нотации, которая получила название "шара и шарнира" или "леденца на палочке". Эта нотация позволяет компактно изобразить графическое соединение компонентов и удовлетворяет требованию масштабирования сложных систем. Предоставляемые интерфейсы компонента обозначаются линией с шаром, а требуемые интерфейсы — линией с гнездом. Так, например, компонент `Заказ` (рис. 11.3) имеет два предоставляемых интерфейса с именами: `Местонахождение Товара` и `Сопровождение`, и два требуемых интерфейса с именами: `Заказываемый Товар` и `Клиент`. В свою очередь компонент `Товар` имеет один предоставляемый интерфейс с именем `Заказываемый Товар` и один требуемый интерфейс с именем `Счет-фактура`.

Наряду с интерфейсами, которые относятся к внешнему представлению, на диаграмме может быть изображено внутреннее представление или представление компонента в форме белого ящика. В этом случае внутри символа ком-

понента изображаются классы или другие классификаторы, которые реализуют поведение этого компонента. Так, например, компонент `Заказ` имеет два класса, которые реализуют его функциональность, а именно: `ЗаголовокЗаказа` и `СтрокаТовара` (см. рис. 11.3). Причем эти классы связаны отношением композиции.

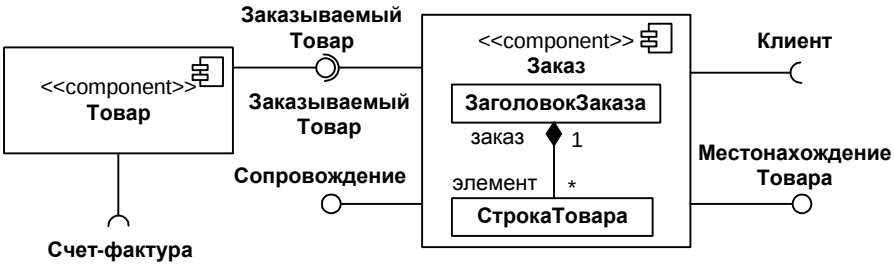


Рис. 11.3. Пример изображения компонентов с интерфейсами в нотации "леденца на палочке"

Для отображения полной сигнатуры интерфейсов компонента с целью показать детали операций, интерфейсы также могут быть изображены в форме обычных прямоугольников класса со стереотипом `<<interface>>` и секцией операций. В этом случае требуемый интерфейс обозначается посредством отношения зависимости, направленной от компонента или от одного из входящих в его состав классов к этому интерфейсу. Дополнительно это отношение может иметь стереотип `<<use>>` (рис. 11.4, *слева*). Предоставляемый интерфейс обозначается с помощью отношения реализации, направленного от компонента или от одного из входящих в его состав классов к этому интерфейсу (рис. 11.4, *справа*). Нетрудно заметить, что эта нотация практически идентична используемой на диаграмме классов.

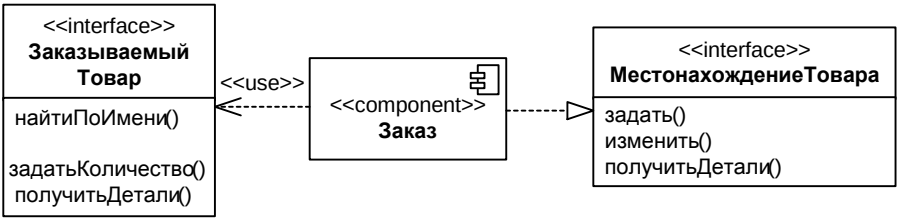


Рис. 11.4. Представление интерфейсов в форме символа классификатора с отношениями зависимости и реализации

Требуемые и предоставляемые интерфейсы компонента принимают во внимание спецификацию структурных характеристик, таких как атрибуты и кон-

цы ассоциации, а также как характеристик поведения, таких как операции и события. Требуемые и предоставляемые интерфейсы необязательно могут быть организованы через порты, это обеспечивает определение именованных множеств предоставляемых и требуемых интерфейсов, к которым обычно (но не всегда) обращаются во время выполнения.

Компонент является заменяемой единицей, которая может быть заменена во время проектирования или во время выполнения компонентом, который предлагает эквивалентную функциональность, основанную на совместимости его интерфейсов. До тех пор, пока окружение удовлетворяет ограничениям, выраженным с помощью предоставляемых и требуемых интерфейсов компонента, он будет способен взаимодействовать с этим окружением.

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие ее части. Таким образом, назначение интерфейсов существенно шире, чем спецификация взаимодействия с пользователями системы или обеспечение взаимодействия компонентов.

Предоставляемые интерфейсы в ранних работах по моделированию программных систем называют также *экспортируемыми*, а требуемые интерфейсы — *импортируемыми*. Хотя эта терминология сейчас практически не употребляется, она может представлять интерес для разработчиков в историческом контексте.

Связи интерфейсов с компонентами могут быть организованы через порты, которые описываются в следующем разделе.

11.4. Порт

Порт, как элемент модели структуры класса, уже рассматривался ранее в *главе 5*. Поэтому здесь будут отмечены те особенности порта, которые характерны для их представления на диаграммах компонентов. Порт определяет различимую точку взаимодействия между компонентом и окружающей его средой или между компонентом и его внутренними частями. Изображаются порты так же, как и на других диаграммах структуры — в виде небольших квадратов, расположенных на границе символа компонента (рис. 11.5). Наличие имени у порта не является обязательным. При отсутствии имени порта его тип ассоциируется с типом интерфейса, с которым связан порт.

Особенности компонентной разработки программ состоят в том, что конкретные компоненты могут по-разному соединяться между собой, в результате чего могут создаваться различные конечные продукты. Это напоминает

игрушечный конструктор. Возможность реконфигурирования компонентов обуславливается тем, что каждый компонент имеет свой собственный адаптер, который осуществляет преобразование любых сообщений или сигналов в язык или форму, понятную другим компонентам. Такие встроенные адаптеры компонентов и получили название портов.

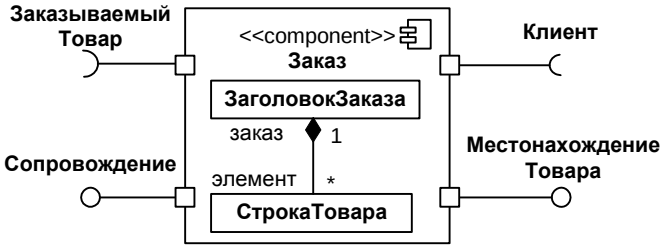


Рис. 11.5. Примеры изображения портов компонентов

Если взглянуть на порты изнутри компонента, то их можно разделить на две разновидности: передающие и конечные порты. *Передающие* порты предназначены для соединения с другими компонентами также посредством портов и служат для транзитной передачи сообщений и сигналов. *Конечные* порты, как правило, предназначены для соединения с конечным автоматом компонента с целью моделирования источников и приемников сообщений и сигналов.

Каждый компонент должен быть достаточно хорошо специфицирован, чтобы он мог использоваться без необходимости описания его внутреннего устройства. Спецификация компонента должна определять, каким образом могут быть установлены все соединения компонента и к какому виду они относятся. Эти определения должны быть стандартизированы еще до того, как будут реализованы сами компоненты. Впоследствии из компонентов могут быть скомпонованы различные приложения, но без спецификации соединений и портов компонентов компонентно-ориентированная разработка становится невозможной.

Порт может быть реализован как отдельный объект со своими собственными свойствами. При задании портов могут быть дополнительно определены типы передаваемых параметров, протокол взаимодействия и язык, при помощи которого может быть выражен протокол. Протокол взаимодействия может быть представлен в форме асинхронного вызова операции, широковещательной рассылки сигнала, буферизированного хранения карты сообщений, специальных протоколов передачи файлов или потоков данных.

Если нотация объединения соединителей используется для связи сложных портов, которые одновременно связаны с предоставляемыми и требуемыми интерфейсами, то для уточнения вида интерфейса на диаграмме компонентов они могут быть дополнительно помечены ключевыми словами `{provided}` или `{required}`.

Порты соединяются с другими портами через соединители, посредством которых могут быть сделаны запросы для активизации поведения компонента.

11.5. Соединитель

Соединитель представляет собой некоторую связь, которая обеспечивает коммуникацию между двумя и более компонентами. Различают два вида соединителей: собирающие соединители и делегирующие соединители.

11.5.1. Собирающий соединитель

Собирающий соединитель (assembly connector) — соединитель, который связывает два компонента в контексте предоставляемых и требуемых сервисов.

Собирающий соединитель предназначен для связи требуемых интерфейсов или портов с предоставляемыми интерфейсами или портами. Собирающий соединитель соединяет два компонента, один из которых предоставляет некоторый сервис или поведение, а другой — нуждается в этом сервисе или поведении. Дополнительно собирающий соединитель имеет направление от требуемого интерфейса или порта к предоставляемому интерфейсу или порту.

Собирающий соединитель обозначается посредством соединения между предоставляемым и требуемым интерфейсами в форме нотации "шара и шарнира". Ранее рассмотренный фрагмент диаграммы компонентов (см. рис. 11.3) может служить примерами графического изображения собирающего соединителя. В качестве другого примера собирающего соединителя можно привести определение компонентов в некоторой библиотеке (рис. 11.6, *а*). Сборка этих компонентов для конкретного приложения может быть представлена в форме следующего фрагмента диаграммы компонентов (рис. 11.6, *б*).

В том случае, когда несколько компонентов предоставляют или требуют один и тот же интерфейс, может быть использована нотация объединения нескольких соединителей, аналогично отношениям обобщения и агрегации. В качестве примера этой нотации можно привести диаграмму компонентов для сборки некоторого приложения (рис. 11.7).

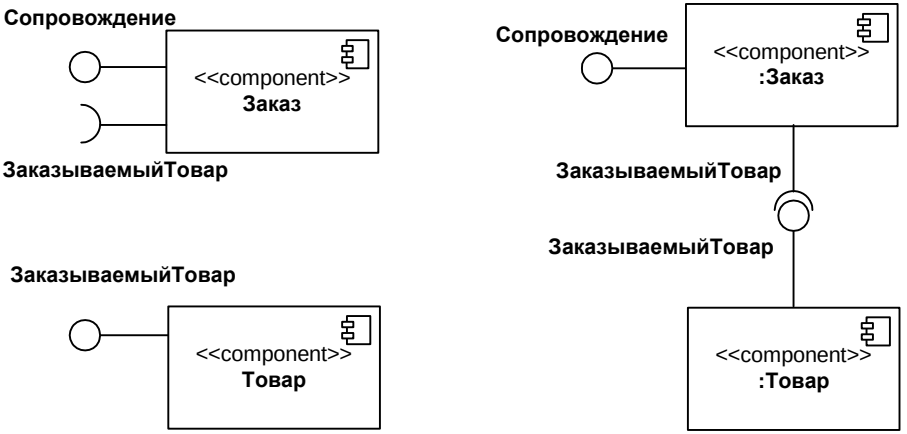


Рис. 11.6. Собирающий соединитель: *а* — в библиотеке компонентов; *б* — в конкретной сборке

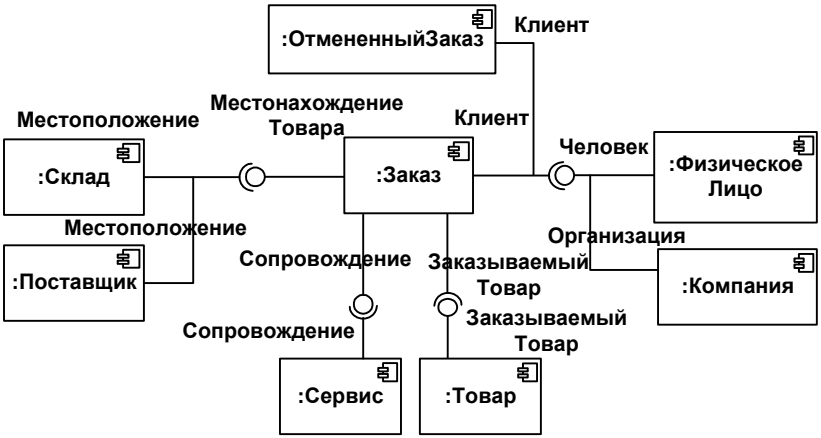


Рис. 11.7. Пример диаграммы компонентов с собирающими соединителями для одинаковых интерфейсов

Совместимость предоставляемых и требуемых интерфейсов при их соединении обеспечивает то, что существующий компонент в системе может быть заменен другим, который предлагает такое же множество сервисов. В контексте, когда компоненты используются для расширения системы с помощью предоставления существующих сервисов, но с добавлением новой функцио-

нальности, собирающие соединители также могут быть использованы для связи с новым определением компонента. Это может быть выполнено посредством добавления нового типа компонента, который предлагает то же множество сервисов, что и существующие типы, и определения новых собирающих соединителей для связи его предоставляемых и требуемых интерфейсов с существующими интерфейсами и портами в некоторой сборке.

Для собирающего соединителя дополнительно определяется семантика времени выполнения. Суть ее заключается в том, что сигналы следуют вдоль экземпляра соединителя, отправляясь от порта с требуемым интерфейсом и направляясь к порту предоставляемого интерфейса. Наличие нескольких соединителей, направленных из одного требуемого интерфейса или порта к предоставляемым интерфейсам двух или более других компонентов, указывает на то, что экземпляр, который должен обработать этот сигнал, будет определен во время выполнения.

Аналогично, наличие нескольких требуемых интерфейсов от различных компонентов, соединенных вместе с одним предоставляемым интерфейсом, указывает на то, что запрос или сигнал должен быть отправлен от экземпляра компонента, который будет определен во время выполнения.

11.5.2. Делегирующий соединитель

Делегирующий соединитель (delegation connector) — соединитель, который связывает внешний контракт компонента с реализацией этого поведения внутренними частями этого компонента.

Делегирующий соединитель связывает порт компонента с внутренними частями этого компонента и предназначен для пересылки им или от них сигналов и сообщений. Сигнал, достигающий порта с делегирующим соединителем, будет передан далее соответствующей внутренней части или другому порту для обработки.

В общем случае делегирующий соединитель выполняет одну из следующих задач:

- передача сообщений или сигналов, поступающих в порт компонента извне, для обработки в некоторую внутреннюю часть компонента или другой порт;
- передача сообщений или сигналов, поступающих из некоторой внутренней части компонента, для обработки во внешний порт компонента.

Смысл делегирования состоит в том, что некоторое поведение не может быть непосредственно реализовано самим экземпляром компонента, и поэтому право реализации передается другому компоненту или классу.

Делегирующий соединитель обозначается в форме линии со стрелкой, направленной или от порта с предоставляемым интерфейсом к некоторой внутренней части компонента (рис. 11.8, *слева*), или от некоторой внутренней части компонента к порту с требуемым интерфейсом (рис. 11.8, *справа*). Например, компонент `Заказ` имеет два делегирующих соединителя для предоставляемого интерфейса `Местонахождение Товара`, направленных от порта к частям компонента с именами `ЗаголовокЗаказа` и `СтрокаТовара`, и один делегирующий соединитель для требуемого интерфейса `Клиент`, направленный от части с именем `СтрокаТовара` к соответствующему порту.

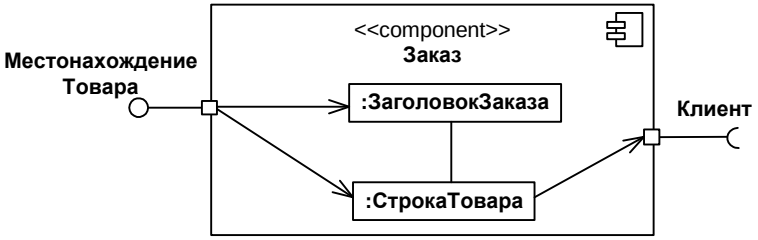


Рис. 11.8. Примеры изображения делегирующих соединителей

Делегирующий соединитель используется в контексте спецификации поведения, которое является доступным для клиентов этого компонента, но в действительности не реализуется самим этим компонентом. Реализовывать это поведение может либо другой компонент, либо отдельный класс. В последнем случае делегирующий соединитель связывает порт компонента с некоторым классом, который содержится в этом компоненте в качестве его части.

Делегирующие соединители могут быть использованы для моделирования иерархической декомпозиции поведения, когда сервисы, предоставляемые компонентом, могут быть реализованы его частью, которая является вложенной в него на несколько уровней глубины. Наличие делегирующего соединителя предполагает, что между соединенными портами будет иметь место поток конкретных сообщений и сигналов даже на несколько уровнях.

Следует заметить, что такой поток сигналов может быть не всегда реализуем в любом окружении или реализации системы, т. е. он может только предполагаться во время проектирования системы.

Порт компонента может быть связан делегирующим соединителем с другими портами вложенных компонентов, которые должны совместно предоставлять функциональность компонента. Во время выполнения сигналы будут направляться соответствующему порту компонента контейнера и далее его внут-

ренним компонентам. В случае, когда несколько портов вложенных компонентов поддерживают обработку одного и того же сигнала, этот сигнал будет направлен всем этим портам.

Если требуется представить компонент в форме представления белого ящика на уровне экземпляра, то для этого компонента может быть определена внутренняя структура, состоящая из частей и делегирующих соединителей (рис. 11.9). Это позволяет явно показать имена частей или имена соединителей в ситуациях, когда одна и та же ассоциация является типом более чем одного соединителя. Такая ассоциация может инстанцироваться внутри компонента несколько раз, играя различные роли в его реализации. В этом случае для делегирующих соединителей может дополнительно использоваться стереотип `<<delegate>>`.

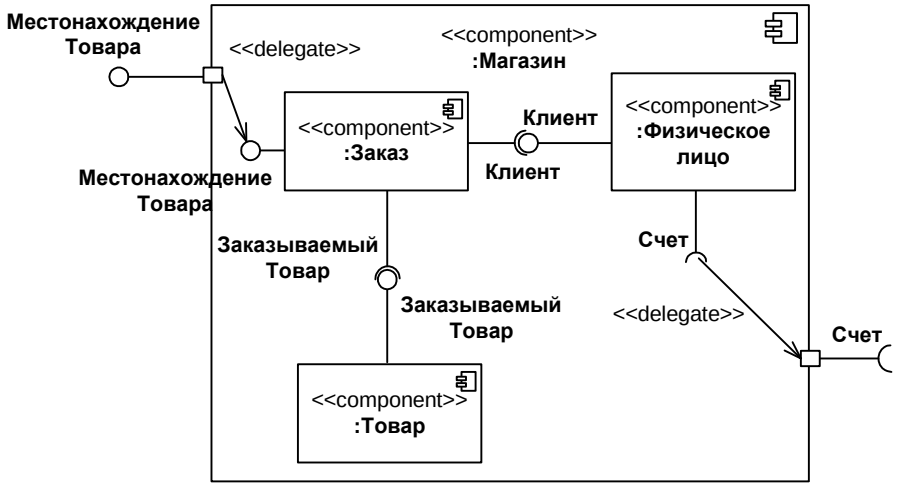


Рис. 11.9. Пример внутренней структуры экземпляра компонента, который содержит другие компоненты как свои части

При использовании делегирующих соединителей в модели необходимо помнить о следующих ограничениях.

- Делегирующий соединитель может быть определен только между интерфейсами или портами одного и того же вида. А именно, или между предоставляемым интерфейсом и портом, или между требуемым интерфейсом и портом.
- Если делегирующий соединитель определяется между требуемым интерфейсом (портом) и некоторой внутренней частью, то эта внутренняя часть должна иметь отношение реализации с интерфейсом (портом) этого типа.

- Если делегирующий соединитель определяется между интерфейсом (портом) в качестве источника и интерфейсом (портом) в качестве цели, то интерфейс в качестве цели должен поддерживать сигнатуру, совместимую с подмножеством операций интерфейса (порта) источника.

На практике соединителю может соответствовать некоторый протокол, о котором условились проектировщики портов. В языке UML 2.0 количество видов соединителей сведено к минимуму. Однако разработчики могут специфицировать дополнительные разновидности соединителей с целью увеличить возможности графического представления связанных портов. При этом возможна простая реализация этой связи в форме указателя или сложная реализация в форме некоторого сетевого соединения.

Внутренние части компонента, такие как классы или другие вложенные компоненты, могут быть показаны явно, как элементы, изображенные внутри символа компонента. Однако эта нотация не является единственной, поскольку они могут быть изображены альтернативно с использованием отношения зависимости.

11.6. Зависимость

Отношение зависимости уже рассматривалось ранее в *главах 3* и *5*. Напомним, что отношение зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Напомним также, что отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента или зависимого элемента к источнику или независимому элементу.

На диаграмме компонентов также могут быть представлены отношения зависимости между компонентами. Это представление часто называют обзорным, поскольку на нем не изображаются интерфейсы и порты. Так, например, рассмотренная ранее диаграмма компонентов с собирающими соединителями для одинаковых интерфейсов (см. рис. 11.7) может быть представлена обзорно с помощью отношений зависимости между компонентами (рис. 11.10). Как нетрудно заметить, в этом случае предоставляемый интерфейс является источником зависимости, а требуемый интерфейс является клиентом зависимости.

В одних случаях зависимости между компонентами могут интерпретироваться как связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода. В других случаях это отношение может характеризо-

вать наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов.

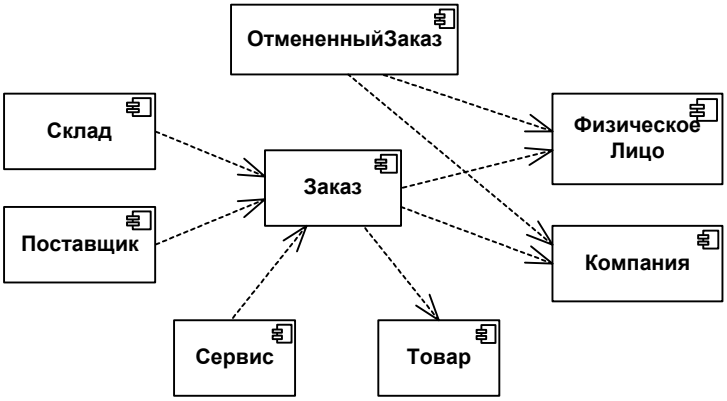


Рис. 11.10. Пример отношений зависимости между компонентом

На диаграмме компонентов зависимости могут также связывать интерфейсы компонентов. Эта нотация может использоваться, например, для спецификации того факта, что компоненты, представленные на диаграмме, определены в разных библиотеках. В качестве примера можно модифицировать рассмотренный ранее фрагмент диаграммы компонентов (см. рис. 11.7). Соответствующий вариант диаграммы компонентов с зависимостями между интерфейсами изображен на рис. 11.11.

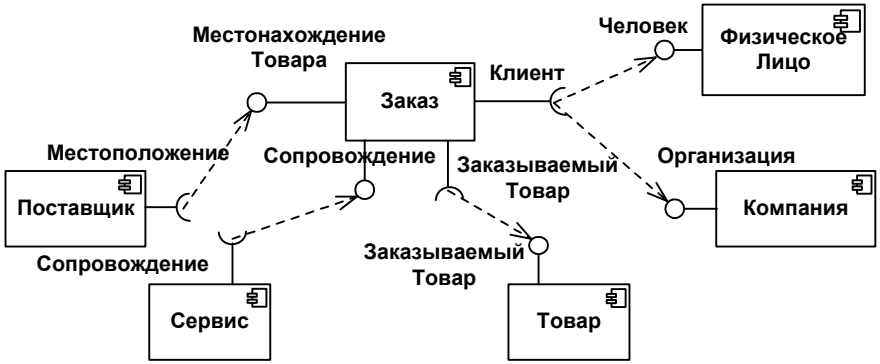


Рис. 11.11. Отношения зависимости на диаграмме компонентов с интерфейсами

Наконец, отношения между компонентами в некоторой системе могут быть структурно определены с использованием специального вида зависимости — отношения реализации.

11.7. Реализация

Реализация (realization) — специализация отношения зависимости для связи компонентов с классификаторами, которые реализуют функциональность этого компонента.

В общем случае поведение компонента может быть реализовано одним или несколькими классами. Как следствие, компонент может выступать в роли некоторой абстракции для коллекции соответствующих элементов модели. При этом считается, что компонент связан отношением реализации с этими элементами. Внутренние классы, которые реализуют поведение компонента, могут быть изображены либо в форме изображения этих классов внутри символа компонента, либо посредством отношения реализации компонента.

Реализация компонента обозначается тем же способом, что и общее отношение зависимости, — в форме пунктирной линии с "V"-образной стрелкой, которая направлена к символу компонента. На противоположном от стрелки конце располагается элемент, который предоставляет реализацию компонента. Так, например, компонент `Заказ`, который содержит два класса, реализующих его функциональность (см. рис. 11.5), может быть представлен в форме отношения реализации следующим образом (рис. 11.12). При этом реализация компонента может быть дополнительно помечена стереотипом `<<implement>>`, который размещается рядом с пунктирной линией зависимости.

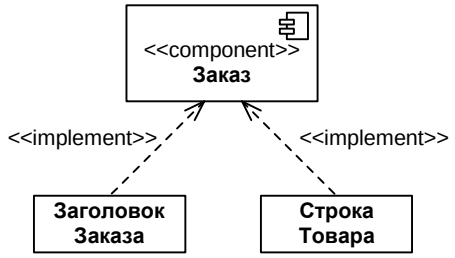


Рис. 11.12. Пример отношения реализации компонента

Если для спецификации отдельного компонента требуется множество различных реализаций, то для большей наглядности можно использовать определенные в языке UML 2.0 стандартные стереотипы компонентов и классов: `<<specification>>` и `<<realization>>`.

Следует заметить, что рассмотренная интерпретация реализации компонентов отличается от представления некоторых инструментальных средств, в которых принято считать, что компонент реализует классы. Именно в случаях неоднозначной интерпретации реализации компонента оказываются полезными указанные выше два стереотипа.

11.8. Стереотипы компонентов

Компонент, как элемент модели, может иметь различную физическую реализацию. Для более точной спецификации формы реализации компонента можно использовать *текстовый* стереотип, который указывается выше строки с именем компонента. В языке UML 2.0 и работах различных авторов для компонентов определено множество текстовых стереотипов, наиболее распространенные из которых представлены в табл.11.1.

Таблица 11.1. *Основные текстовые стереотипы компонентов*

Ключевое слово	Перевод	Назначение
`file`	Файл	Определяет наиболее общую разновидность компонента, который представляется в виде произвольного физического файла
`executable`	Исполнимый	Определяет разновидность компонента-файла, который является исполнимым файлом и может выполняться на некоторой компьютерной платформе
`document`	Документ	Определяет разновидность компонента-файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы
`library`	Библиотека	Определяет разновидность компонента-файла, который представляется в форме динамической или статической библиотеки
`source`	Источник	Определяет разновидность компонента-файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл
`table`	Таблица	Определяет разновидность компонента, который представляется в форме таблицы базы данных

Для более наглядного изображения компонентов также были предложены и реализованы во многих инструментальных средствах *графические* стереотипы, которые представляют собой специальные графические символы, иллюстрирующие конкретную форму или особенности реализации компонента. Хотя эти дополнительные обозначения не специфицированы в нотации языка UML 2.0, однако, удовлетворяя общим механизмам расширения языка UML, они упрощают понимание диаграммы компонентов, существенно повышая наглядность ее графического представления.

Из графических стереотипов наибольшее распространение получили три группы символов. Первая из них, которая впервые встречается в работах Г. Буча, содержит стереотипы для общих компонентов или артефактов развертывания. Для таких компонентов используются следующие стереотипы:

- динамически подключаемые библиотеки, как правило имеющие расширение DLL (рис. 11.13, *а*);
- Web-страницы на языке разметки гипертекста, как правило, имеющие расширение HTML или HTML (рис. 11.13, *б*);
- файлы справки, как правило, имеющие расширение HLP (рис. 11.13, *в*);
- файлы с исходными текстами программ, имеющие, например, расширения H и CPP для языка C++ (рис. 11.13, *г*).

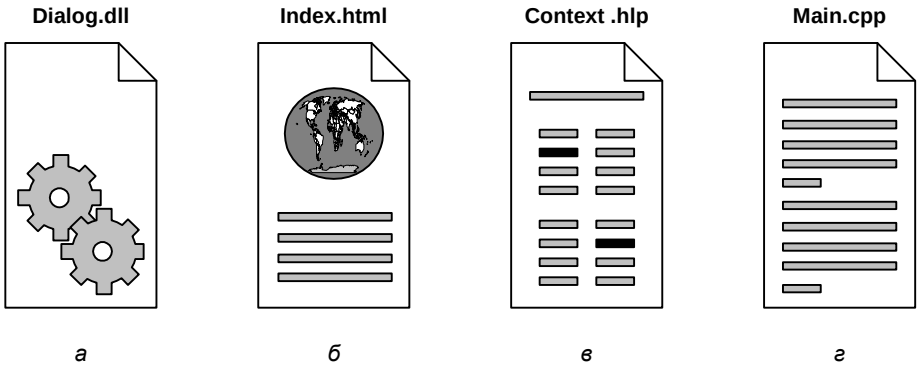


Рис. 11.13. Изображение графических стереотипов компонентов Г. Буча: *а* — динамически подключаемых библиотек; *б* — Web-страниц; *в* — файлов справки; *г* — файлов с исходными текстами программ

Отдельные разработчики предложили собственные графические стереотипы для изображения тех или иных разновидностей компонентов. Среди них наибольшее применение получили стереотипы Дж. Коналлена, которые ориентированы для представления диаграмм компонентов Web-приложений.

Эти графические стереотипы образуют вторую группу, основные из стереотипов которой представлены в табл. 11.2. Следует заметить, что этот набор графических стереотипов реализован в CASE-средстве MagicDraw UML 10.0 (No Magic, Inc).

Таблица 11.2. *Графические стереотипы компонентов Дж. Коналлена*

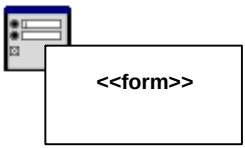
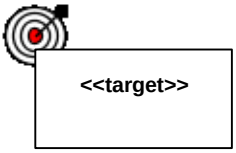
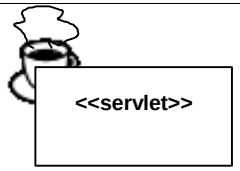
Графический символ	Назначение
	Серверная страница представляет Web-страницу, содержащую выполняемые сервером сценарии. Эти сценарии могут взаимодействовать с серверными ресурсами, такими как базы данных, бизнес-логика и внешние системы. Операции реализуемых компонентов классов являются функциями сценария, а их атрибуты — переменными, видимыми в пределах этой страницы
	Клиентская страница представляет Web-страницу в формате HTML, а также данные, элементы интерфейса и даже бизнес-логику. Клиентские страницы отображаются клиентскими браузерами и могут содержать сценарии, которые интерпретируются браузером. Операции клиентской страницы могут соответствовать функциям, содержащимся в дескрипторах сценария страницы. Атрибутам клиентской страницы соответствуют объявленные в дескрипторах сценария переменные, которые доступны любой функции в пределах этой страницы
	Форма является набором полей ввода и представляет собой часть клиентской страницы. Форма преобразуется непосредственно в дескриптор HTML `<form>`. Атрибуты формы могут представлять поля ввода, текстовые поля, переключатели, флажки, скрытые поля формы HTML. С формой не связано никаких операций, поскольку их нельзя в ней инкапсулировать. Любые операции взаимодействия с формой являются свойствами содержащей ее страницы
	Набор фреймов представляет собой контейнер, состоящий из нескольких Web-страниц. Прямоугольная область просмотра делится на несколько фреймов. Каждый фрейм может быть связан с одним объектом со стереотипом `<<target>>`, однако это необязательно. Содержимым фрейма может быть Web-страница или другой фрейм. Набор фреймов преобразуется непосредственно в набор фреймов Web-страницы и дескриптор HTML `<frame>`. Поскольку на практике набор фреймов является клиентской страницей, этот компонент также может включать классы с операциями и атрибутами, которые могут активизироваться лишь теми браузерами, которые не отображают фреймы

Таблица 11.2 *(окончание)*

Графический символ	Назначение
	Цель представляет собой именованную область окна браузера, в которой могут отображаться Web-страницы. Имя цели соответствует имени целевого объекта. Обычно целью является один из фреймов набора. Однако целью может быть и новое окно браузера. Для цели может быть задано место назначения, где будет отображена новая Web-страница. Имя цели должно быть уникальным для каждого клиента системы. Таким образом, для каждого клиента может существовать лишь один экземпляр цели
	Этот компонент представляет Web-страницу, которую браузер может запрашивать по ее имени. Этот компонент при необходимости может содержать клиентские или серверные сценарии. Обычно Web-страницы являются текстовыми файлами, доступ к которым можно получить через Web-сервер. Однако они могут быть также компилируемыми модулями, загружаемыми и запускаемыми Web-сервером. В любом случае при доступе к такой странице, хранящейся в файле или исполняемой Web-сервером, она генерирует документ в формате HTML, который отправляется в ответ на запрос браузера
	Этот компонент представляет Web-страницы, реализующие код JSP серверной части приложения. Данный стереотип применим лишь к приложениям, в которых используется технология Java Server Pages
	Этот компонент представляет сервлет Java. Стереотип применим лишь к приложениям, поддерживающим сервлеты компании Sun

Некоторые инструментальные CASE-средства, прежде всего, это относится к IBM Rational Rose, содержат самостоятельный набор графических символов для обозначения компонентов. К сожалению, объем и тематика книги не позволяет детально остановиться на этих стереотипах, которые образуют третью группу. За дополнительной информацией по этим вопросам можно обратиться к одной из предыдущих книг автора, которые приводятся в списке литературы.

Если рассмотренных стереотипов оказывается недостаточно, то разработчики могут использовать в качестве графических стереотипов самостоятельные обозначения. Важно помнить, что хотя в языке UML 2.0 нет строгой нотации для изображения графических стереотипов компонентов,

предложенные разработчиками обозначения не должны повторять или переопределять уже известные стереотипы.

Следует отметить, что на практике диаграмма компонентов строится только в том случае, если предполагается программная реализация разрабатываемой модели. В противном случае данный тип канонических диаграмм может вообще отсутствовать в модели, что характерно для проектов документирования и реинжиниринга бизнес-процессов.

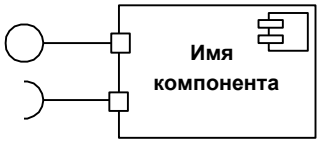
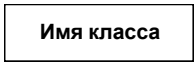
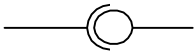
* * *

В заключение приводится сводка основных графических обозначений, которые могут быть использованы при построении диаграмм компонентов (табл. 11.3).

Таблица 11.3. *Графические узлы и пути на диаграммах компонентов*

Тип графического элемента	Нотация	Ссылка на раздел
Компонент (component) с текстовым стереотипом	<<component>> **Имя компонента**	Компонент *(см. разд. 11.2)*
Компонент (component) с пиктограммой стереотипа	**Имя компонента**	Компонент *(см. разд. 11.2)*
Компонент с предоставляемым интерфейсом (provided interface)	**Имя компонента**	Интерфейс *(см. разд. 11.3)*
Компонент имеет порт (port) с предоставляемым интерфейсом	**Имя компонента**	Порт *(см. разд. 11.4)*
Компонент с требуемым интерфейсом (required interface)	**Имя компонента**	Интерфейс *(см. разд. 11.3)*
Компонент имеет порт (port) с требуемым интерфейсом	**Имя компонента**	Порт *(см. разд. 11.4)*

Таблица 11.3 *(окончание)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Компонент имеет несколько портов (ports) с предоставляемыми и требуемыми интерфейсами		Порт *(см. разд. 11.4)*
Класс (class)		Класс *(см. разд. 4.2)*
Часть (part)		Часть *(см. разд. 5.2.1)*
Собирающий соединитель (assembly connector)		Собирающий соединитель *(см. разд. 11.5.1)*

Глава 12

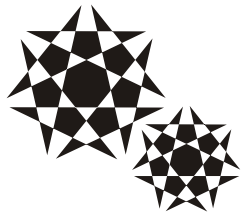


Диаграмма развертывания (deployment diagram)

Физическое представление модели программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Конечно, если разрабатывается простое приложение, которое может выполняться локально на компьютере пользователя, не используя никаких распределенных устройств и сетевых ресурсов, то в этом случае нет необходимости в разработке дополнительных диаграмм физического уровня представления. Однако при разработке сложных корпоративных или распределенных приложений ситуация представляется совсем иначе.

Во-первых, сложные программные системы могут реализовываться в сетевом варианте, используя различные вычислительные платформы и технологии доступа к базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач по рациональному размещению компонентов по узлам этой сети, что оказывает непосредственное влияние на общую производительность программной системы.

Во-вторых, интеграция программной системы с глобальной сетью Интернет приводит к необходимости решения дополнительных задач при проектировании системы. Такими задачами, прежде всего, является обеспечение безопасности, криптозащищенности и устойчивости доступа к информации для корпоративных клиентов. Успешное решение этих задач в немалой степени зависит от проекта реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, межсетевые экраны, каналы связи и хранилища данных.

Наконец, технологии доступа и манипулирования данными в рамках общей схемы "клиент-сервер" также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования,

кэширования для обеспечения необходимой производительности и устойчивости системы. Все эти аспекты требуют визуального представления модели распределенной системы и ее топологии с целью спецификации программных и аппаратных особенностей реализации сетевых архитектур.

Как было отмечено в *главе 11*, первой из диаграмм физического представления является диаграмма компонентов. Второй формой физического представления программной системы является диаграмма развертывания (синоним — диаграмма размещения).

Диаграмма развертывания (deployment diagram) предназначена для представления общей конфигурации или топологии распределенной программной системы и содержит изображение размещения различных артефактов по отдельным узлам системы.

Кроме этого, диаграмма развертывания показывает наличие физических соединений или маршрутов для передачи информации между аппаратными и программными устройствами, которые обеспечивают функционирование системы в различных режимах.

При разработке диаграмм развертывания преследуются следующие цели:

- ❒ специфицировать физические узлы, необходимые для размещения на них исполнимых компонентов программной системы;
- ❒ показать физические связи между узлами реализации системы на этапе ее исполнения;
- ❒ выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для успешного достижения этих целей для разработки диаграмм развертывания привлекаются системные аналитики, сетевые инженеры и системотехниками.

Диаграмма развертывания визуализирует только те элементы физического представления модели, которые существуют во время выполнения или исполнения (run-time) программной системы. Такими элементами могут быть конкретные артефакты, такие как исполнимые файлы и библиотеки динамической компоновки. Те артефакты, которые не используются на этапе выполнения, на диаграмме развертывания, как правило, не показываются. Так, например, компоненты с исходными текстами программ могут присутствовать на диаграмме компонентов, а на диаграмме развертывания они не указываются.

Диаграмма развертывания может содержать графические изображения процессоров, периферийных устройств, артефактов и различных отношений

между ними. В отличие от диаграмм логического представления, диаграмма развертывания является, как правило, единственной для разрабатываемой системы, поскольку должна всецело отражать особенности ее топологии и реализации. Эта диаграмма, по сути, завершает процесс ООАП для конкретной программной системы, а ее разработка является последним этапом спецификации модели.

В языке UML 2.0 специфицирована некоторая упрощенная модель развертывания, которая оказывается достаточной для большинства практических приложений. Для более детального представления моделей распределенных систем эта концепция развертывания может быть расширена с помощью дополнительных профилей, которые уточняют семантику специальных аппаратных устройств, платформ и сред программного обеспечения.

12.1. Узел

Узел (node) является элементом модели, который представляет некоторый вычислительный ресурс для развертывания на нем различных артефактов.

На диаграммах развертывания узлы обычно служат для представления аппаратных устройств или исполняемых программных средств. С каждым узлом ассоциируется некоторое множество элементов, которые развертываются на нем. Развернутые на узлах элементы представляют собой артефакты.

На диаграммах развертывания узел изображается в форме фигуры, которая представляет собой двумерное изображение трехмерного куба или прямоугольного параллелепипеда. Узел имеет имя, которое указывается внутри этого графического символа. На диаграммах развертывания узлы могут представляться или в качестве типа (рис. 12.1, *а*), или в качестве экземпляра (рис. 12.1, *б*).

Рис. 12.1. Графическое изображение: *а* — узла в качестве типа; *б* — узла в качестве экземпляра

В первом случае имя узла записывается в форме обычного имени классификатора: <*имя-типа-узла*>. При этом имя начинается с прописной буквы,

а строка имени не подчеркивается. Имя типа узла указывает на некоторую разновидность узлов, присутствующих в модели системы.

Во втором случае имя экземпляра узла записывается в следующем формате (БНФ):

<имя-экземпляра-узла>::=[<собственное-имя-узла>][':'<имя-типа-узла>]

При этом собственное имя узла записывается со строчной буквы, а вся запись подчеркивается.

Для моделирования низкоуровневых приложений узлы могут иметь внутреннюю структуру, определенную в терминах частей и соединителей. Если определена внутренняя структура узла, то она может содержать только части типа "узел". Иерархически вложенные узлы можно моделировать с использованием композиций.

Для представления топологии сетей и моделей сетевых архитектур произвольной сложности на диаграмме развертывания узлы могут быть соединены между собой путями коммуникации. Пути коммуникации определяются между такими узлами, как, например, сервер приложений и клиентские рабочие станции. Пути коммуникации между экземплярами узлов могут специфицировать конкретные варианты сетей со специальной топологией.

На практике для уточнения спецификации узла могут использоваться различные текстовые стереотипы, которые акцентируют внимание на назначении этого узла. Хотя в языке UML 2.0 конкретные стереотипы для узлов не определены, разработчики предложили для этой цели следующие текстовые стереотипы: `<<application server>>` (сервер приложений), `<<client workstation>>` (клиентская рабочая станция), `<<mobile device>>` (мобильное устройство), `<<embedded device>>` (встроенное устройство), `<<processor>>` (процессор), `<<sensor>>` (датчик), `<<modem>>` (модем), `<<net>>` (сеть), `<<printer>>` (принтер) и др. Эти стереотипы, как правило, интерпретируются, исходя из физического контекста их применения.

Узлы обычно представляют или среду выполнения программного обеспечения, или аппаратные устройства. Поэтому в языке UML 2.0 рассматриваются узлы двух разновидностей: среда выполнения и устройство.

12.1.1. Среда выполнения

Среда выполнения (execution environment) представляет собой узел, который обладает функциональностью, необходимой для практического выполнения развернутых на нем исполнимых артефактов.

Среда выполнения моделирует программно-аппаратную платформу, предоставляющую набор стандартных услуг, которые могут использоваться исполнимыми компонентами приложений. На этой платформе среда выполнения реализует стандартное множество сервисов, которые требуются артефактам во время своего исполнения. Однако на диаграмме развертывания эти сервисы не специфицируются, поскольку их наличие предполагается неявно при указании стандартной среды выполнения. Для каждого развернутого артефакта могут иметь значение различные аспекты этих сервисов, необходимая совокупность которых может определяться в спецификации развертывания для отдельного вида среды выполнения.

Среда выполнения обозначается символом узла с дополнительным стереотипом `<<executionEnvironment>>`. Примерами сред выполнения являются операционные системы, системы управления технологическими потоками и базами данных, виртуальные машины, эмуляторы и интерпретаторы (рис. 12.2, *а*). Среда выполнения, как правило, является частью более общего узла, представляющего физическую или аппаратную платформу, на которой устанавливается или инсталлируется эта среда выполнения. Графически это изображается посредством размещения символа среды выполнения внутри символа соответствующего узла (рис. 12.2, *б*).

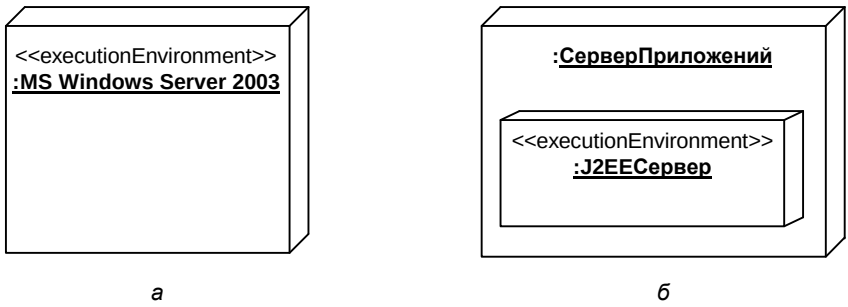


Рис. 12.2. Примеры изображения: *а* — среды выполнения; *б* — среды выполнения в узле

Среды выполнения могут быть вложенными, образуя иерархические структуры, например, среда выполнения базы данных может быть вложенной в среду выполнения операционной системы. В этом случае вложенная среда выполнения играет роль части в другой среде выполнения. На практике среда выполнения формируется посредством развертывания компонентов подходящего типа на специальных узлах.

На практике для дополнительного уточнения среды выполнения могут использоваться различные текстовые стереотипы, такие как: `<<OS>>`,

`<<workflow engine>>`, `<<database system>>` и `<<J2EE container>>`. В случае их применения они записываются после основного стереотипа `<<executionEnvironment>>`, а в качестве символа разделения используется запятая. Например, `<<executionEnvironment, OS>>` или `<<executionEnvironment, database system>>`.

Среда выполнения может иметь необязательные интерфейсы для предоставления сервисов системного уровня, которые могут вызываться развернутыми элементами. Это позволяет разработчику явно специфицировать используемые сервисы среды выполнения.

12.1.2. Устройство

Устройство (device) представляет собой узел, который обладает некоторым общим вычислительным ресурсом со способностью обрабатывать развернутые на нем артефакты.

Устройства могут быть простыми, не содержащими вложенных устройств, и сложными, состоящими из других устройств. Устройство может состоять из вложенных элементов, если оно физически декомпозируется на эти элементы. В качестве вычислительного ресурса узла рассматривается наличие одного или нескольких процессоров и некоторого объема оперативной памяти.

Устройство обозначается символом узла с дополнительным стереотипом `<<device>>` (рис. 12.3, *а*). На диаграммах развертывания в нотации UML 2.0 устройства, как правило, выступают в качестве контейнера для среды выполнения (рис. 12.3, *б*).

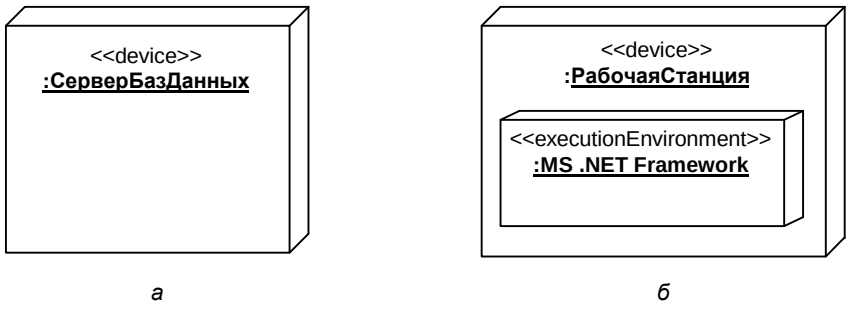


Рис. 12.3. Примеры изображения: *а* — простого устройства; *б* — устройства контейнера

В прежних версиях языка UML 1.*x* понятие устройства использовалось для представления аппаратных, механических или электронных устройств, таких

как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы. Однако на диаграммах развертывания в нотации UML 2.0 для таких устройств более удачным является применение специальных текстовых стереотипов.

Для общего указания узла без конкретизации особенностей его физической реализации оказывается полезным понятие цели развертывания.

Цель развертывания (deployment target) является абстрактным метаклассом для указания местоположения размещаемого артефакта.

Цель развертывания выступает в качестве контейнера для развернутых на ней артефактов. Тем самым обеспечивается согласованный уровень представления артефактов, которые наделяются способностью быть развернутыми на некоторой цели развертывания. Конкретизацией цели развертывания являются различные виды узлов.

Цель развертывания наделяется способностью содержать или владеть некоторым множеством развертываний. Понятие развертывания по сути аналогично слоту композита, в роли которого выступает некоторая цель развертывания. Множество развертываний, которые могут быть размещены в цели развертывания, объявляются в манифесте соответствующего артефакта. Особенности моделирования артефактов рассматриваются в следующем разделе.

12.2. Артефакт

На целях развертывания могут быть развернуты размещаемые артефакты.

Размещаемый артефакт (deployed artifact) является абстрактным метаклассом для представления артефакта или экземпляра артефакта, который должен быть размещен на цели развертывания.

Понятие размещаемого артефакта необходимо для общего указания физической сущности, которая должна быть помещена в некоторое развертывание, расположенное на цели развертывания. При этом размещаемый артефакт может быть помещен в один или более развертываний на цели развертывания.

На практике размещаемые артефакты выступают в роли конкретных физических сущностей, которые являются результатом процесса разработки и реализации программной системы.

Артефакт (artifact) представляет собой элемент модели, который специфицирует некоторую физически существующую часть информации, используемую или производимую в ходе разработки программного обеспечения или в процессе развертывания и функционирования системы.

В общем случае артефакт является классификатором, который представляет некоторую физическую сущность. Артефакты могут иметь свойства, которые представляют характеристики артефакта, и операции, которые могут быть выполнены его экземплярами. Артефакты могут иметь отношения с другими артефактами, например, отношения композиции или зависимости. Для представления семантики конкретной копии артефакта могут быть инстанцированы экземпляры артефакта. При этом различные экземпляры одного и того же артефакта могут быть развернуты на различных экземплярах узла одного типа.

Артефакт представляется с использованием символа прямоугольника классификатора с ключевым словом `<<artifact>>`, записанным в форме текстового стереотипа, и небольшой пиктограммой примечания в правом верхнем углу. На диаграммах развертывания артефакты могут представляться или в качестве типа (рис. 12.4, *а*), или в качестве экземпляра (рис. 12.4, *б*).

Рис. 12.4. Примеры изображения: *а* — артефакта в качестве типа; *б* — экземпляра артефакта

Артефакт имеет имя, которое указывается внутри графического символа. Чтобы иметь возможность ссылаться на артефакт в некотором физическом контексте, в качестве имени используется конкретное имя файла. При этом имя экземпляра артефакта подчеркивается. Никаких дополнительных правил синтаксиса имен артефактов в языке UML 2.0 не указано. Примерами артефактов являются: файлы с исходным кодом, файлы скриптов, исполнимые файлы, файлы архивов и библиотек, таблицы баз данных, разработанные документы или модели, сообщения электронной почты.

Отдельный экземпляр или копия артефакта может быть развернут на экземпляре узла. В этом контексте артефакт связывается с некоторым узлом отношением развертывания. Артефакт также может иметь отношения манифестации с некоторым множеством элементов модели, которые используются при

построении или генерации данного артефакта. Артефакты могут состоять из других артефактов, образуя иерархические структуры. Вложенность артефактов представляется посредством отношения композиции. Эти отношения и примеры их графического изображения рассматриваются далее в этой главе.

Для артефактов определены несколько стандартных стереотипов, например, `<<source>>` и `<<executable>>`. Для уточнения свойств артефактов, реализованных в форме конкретных файлов, эти стереотипы могут быть дополнены стереотипами, основанными на расширениях файлов в некоторой операционной системе или на некоторой вычислительной платформе. Например, в стандартном профиле EJB определен стереотип `<<jar>>` в качестве уточнения стереотипа `<<executable>>` для исполнимых архивов Java. Альтернативно, артефакт может быть изображен специальным графическим стереотипом.

Для более полного описания цели развертывания или размещаемого артефакта оказывается полезным понятие спецификации экземпляра, которое может быть конкретизировано применительно к модели развертывания.

Спецификация экземпляра (instance specification) представляет собой элемент модели, предназначенный для указания свойств конкретной цели развертывания или конкретного размещаемого артефакта.

Спецификация экземпляра может соответствовать некоторой цели развертывания в отношении развертывания, когда цель развертывания соответствует некоторому экземпляру узла. Спецификация экземпляра также может соответствовать некоторому размещаемому артефакту, если цель развертывания является экземпляром артефакта. Спецификация экземпляра связывается с узлом с использованием отношения развертывания. Альтернативной нотацией является изображение символа спецификации экземпляра внутри символа узла.

12.3. Спецификация развертывания

Спецификация развертывания (deployment specification) описывает множество свойств, которые определяют параметры выполнения артефакта компонента, развертываемого на некотором узле.

Спецификация развертывания может быть нацелена на специальный тип контейнера. Артефакт, который материализует или реализует свойства спецификации развертывания является некоторым дескриптором развертывания.

В метамодели спецификация развертывания является подтипом артефакта. Она определяет множество свойств развертывания, которые специализиру-

ются в определенный тип контейнера. Экземпляр спецификации развертывания со специальными значениями для этих свойств может быть помещен в сложный артефакт.

Спецификация развертывания графически изображается в форме прямоугольника классификатора с ключевым словом `<<deployment spec>>`, записанным в форме текстового стереотипа. На диаграммах развертывания спецификации развертывания могут представляться или в качестве типа (рис. 12.5, *а*), или в качестве экземпляра (рис. 12.5, *б*).

а *б*

Рис. 12.5. Примеры изображения спецификации развертывания для артефакта: *а* — уровня типа; *б* — уровня экземпляра

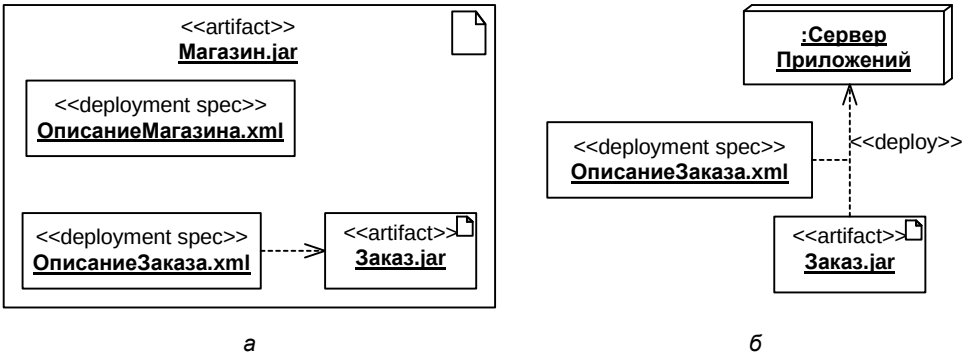
Спецификация развертывания имеет имя, которое указывается внутри графического символа. При этом имя спецификации развертывания для экземпляра артефакта подчеркивается.

В языке UML 2.0 спецификация развертывания является общим механизмом параметризации отношения развертывания. Спецификация развертывания присоединяется к артефакту, размещенному на некотором узле, с использованием обычной нотации зависимости. Так, например, экземпляр артефакта с именем Заказ.jar имеет экземпляр спецификации развертывания с именем ОписаниеЗаказа.xml (рис. 12.6, *а*). Символ спецификации развертывания может быть также изображен вложенным в символ соответствующего артефакта. Так, например, экземпляр артефакта с именем Магазин.jar имеет экземпляр спецификации развертывания с именем ОписаниеМагазина.xml (рис. 12.6, *а*).

Если отношение развертывания изображается в форме нотации зависимости, то символ спецификации развертывания может быть присоединен к этому отношению, аналогично символу примечания (рис. 12.6, *б*).

Как правило, в спецификации развертывания указывается местоположение артефакта при его развертывании на некотором узле. Местоположение может быть локальным или удаленным. На практике таким местоположением обычно является некоторый каталог или адрес памяти. В качестве местоположения может быть также указана среда выполнения, на которой должен выполняться экземпляр артефакта. Последний случай отражает ситуацию,

когда цель развертывания спецификации развертывания является средой выполнения.



а *б*

Рис. 12.6. Примеры графического изображения экземпляров спецификаций развертывания: *а* — вложен в символ соответствующего артефакта; *б* — аналогично символу примечания

Семантика спецификации развертывания может быть расширена в специальных профилях. Для уточнения характеристик спецификаций развертывания артефактов, кроме текстовых стереотипов, могут использоваться также помеченные значения. Тегами таких помеченных значений являются, например, `<<concurrencyMode>>` с возможными значениями `{нить, процесс, нет}` или `<<transactionMode>>` с возможными значениями `{транзакция, вложенная транзакция, нет}`. При использовании помеченных значений они заключаются в фигурные скобки и записываются, как правило, ниже имени соответствующего элемента модели, например, `{concurrencyMode = нить}` для экземпляра артефакта Заказ.jar (рис. 12.7).

12.4. Отношения на диаграмме развертывания

Наиболее общим отношением на диаграмме развертывания является уже известная нам по предыдущим главам зависимость. На диаграммах развертывания отношение зависимости используется для представления либо отношения развертывания артефакта или его экземпляра на узле или его экземпляре, либо для отношения между артефактом и элементом модели, который его реализует. Первый случай характерен для отношения развертывания, а второй — для отношения манифестации. Оба эти отношения рассматриваются далее в этой главе.

12.4.1. Развертывание

Развертывание (deployment) представляет собой размещение артефакта или экземпляра артефакта на некоторой цели развертывания.

В общем случае отношение развертывания представляет размещение одного или более артефактов или экземпляров артефактов на цели развертывания с необязательной спецификацией развертывания. В языке UML 2.0 развертывание является подтипом зависимости. Артефакты, которые развертываются на узле, выступают в роли клиентов зависимости, а узел, который является целью развертывания, выступает в роли источника этой зависимости.

Размещение артефактов на узлах диаграммы развертывания может быть изображено тремя способами. Во-первых, символы экземпляров артефактов могут быть изображены внутри символа экземпляра узла. Это означает, что соответствующие экземпляры артефактов должны быть развернуты на этом экземпляре узла. В качестве примера первого способа изображения можно привести фрагмент диаграммы развертывания для среды выполнения с именем `server1:J2EEСервер`, на которой развернуто 7 экземпляров артефактов (рис. 12.7).

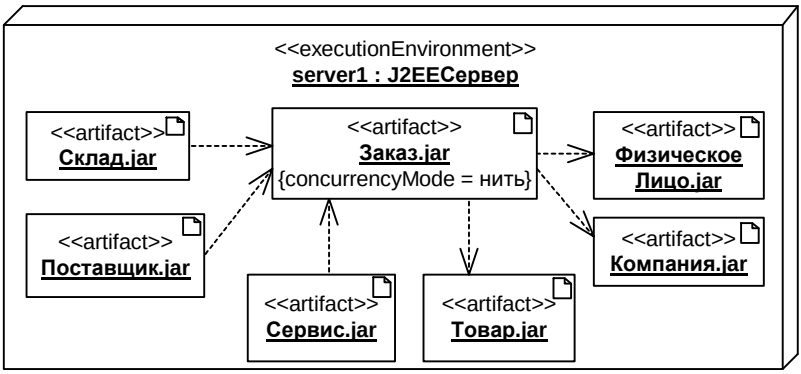


Рис. 12.7. Первый способ визуального представления множества экземпляров артефактов, развернутых на узле

Нетрудно заметить, что этот пример является дальнейшей проработкой модели компонентов, рассмотренной ранее в *главе 11* (см. рис. 11.10). Особенность этого способа изображения состоит в том, что на диаграмме развертывания дополнительно могут быть изображены зависимости между артефактами.

Во-вторых, символ экземпляра артефакта может быть соединен с символом цели развертывания с использованием зависимости, помеченной дополнительно стереотипом <<deploy>>. При этом линии зависимости имеют направление от артефактов к цели развертывания. Зависимость развертывания интерпретируется аналогично первому способу представления, а именно экземпляры артефактов должны быть развернуты на соответствующем экземпляре узла. В качестве примера второго способа изображения можно привести альтернативное представление ранее рассмотренного фрагмента диаграммы развертывания для среды выполнения с именем `server1:J2EEСервер` (рис. 12.8, *а*). В этом примере изображены не все экземпляры артефактов. Как нетрудно заметить, второй способ изображения неудобен для представления большого числа артефактов.

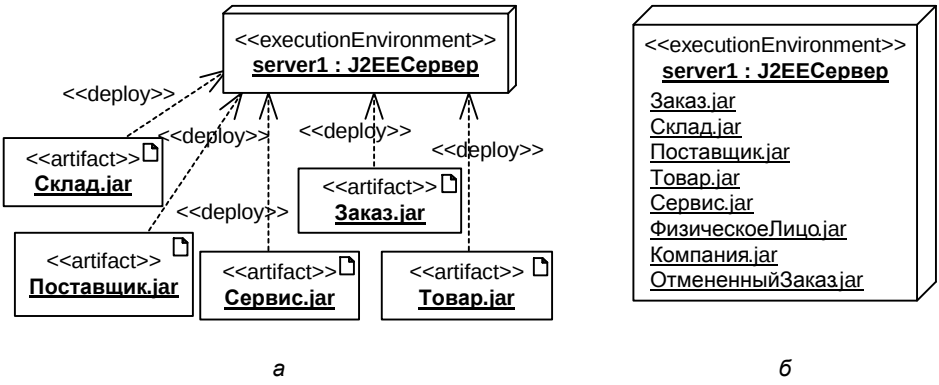


Рис. 12.8. Второй (*а*) и третий (*б*) способы визуального представления множества экземпляров артефактов, развернутых на узле

Наконец, развертывание артефактов может быть изображено в форме списка артефактов внутри символа узла. В качестве примера третьего способа изображения можно привести альтернативное представление ранее рассмотренного фрагмента диаграммы развертывания для среды выполнения с именем `server1:J2EEСервер` (рис. 12.8, *б*). Этим способом могут быть изображены все экземпляры артефактов, поэтому он наиболее удобен при построении моделей развертывания сложных систем, поскольку является самым компактным из всех трех.

В общем случае отношение развертывания между размещаемым артефактом и целью развертывания может быть определено на уровне типов и на уровне экземпляров. Рассмотрим в качестве примера отношение развертывания, которое на уровне типов может быть определено между некоторым узлом

с именем `Сервер приложений` и некоторым исполнимым артефактом с именем `Обработчик запроса`. На уровне экземпляров целями развертывания могут быть, например, 3 экземпляра узлов с именами: `Сервер приложений № 1`, `Сервер приложений № 2` и `Сервер приложений № 3` для 6 экземпляров артефактов с именами: `Обработчик запроса № 1`, ..., `Обработчик запроса № 6`.

Графически изобразить соответствующую диаграмму развертывания для рассмотренного примера предлагается читателям самостоятельно в качестве упражнения.

12.4.2. Манифестация

Следующим отношением, которое используется на диаграммах развертывания, является манифестация.

Манифестация (manifestation) представляет собой отношение для спецификации конкретного физического воплощения одного или нескольких элементов модели посредством артефакта.

Манифестация является специализацией отношения зависимости и находится в собственности артефакта. Артефакт физически реализует один или несколько элементов модели и выступает в роли клиента зависимости. Элементы модели, которыми на диаграммах развертывания являются, как правило, компоненты и классы, выступают в качестве источников зависимости манифестации.

Отношение манифестации обозначается аналогично зависимости в форме пунктирной линии со стрелкой, направленной от артефакта к соответствующему элементу модели. Рядом со стрелкой указывается ключевое слово `<<manifest>>`. Так, например, артефакт с именем Заказ.jar физически реализует компонент с именем `Заказ` (рис. 12.9, *а*). От артефакта может исходить, а к элементу модели могут подходить несколько стрелок манифестации. Так, например, 4 класса с именами `Склад`, `Заказ`, `Товар` и `Сервис` манифестируются одним заголовочным файлом Магазин.h, однако каждому классу соответствует свой файл с исходным кодом на языке C++ (рис. 12.9, *б*).

В общем случае манифестация является отношением типа "многие-ко-многим". Это означает, что один элемент модели может быть реализован несколькими артефактами, а один артефакт может реализовывать несколько элементов модели.

Для отношения манифестации могут быть предложены дополнительные текстовые стереотипы с целью указать отдельные формы манифестации, напри-

мер, <<tool generated>> и <<custom code>>. Эти стереотипы могут быть двумя манифестациями для различных классов, реализованными в некотором артефакте.

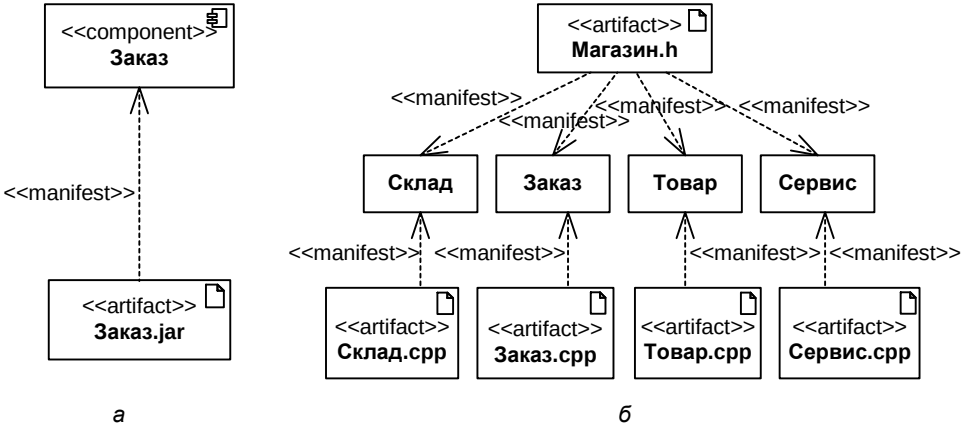


Рис. 12.9. Примеры изображения отношения манифестации:
а — между артефактом и компонентом;
б — артефактами, компонентами и классами

12.4.3. Путь коммуникации

Для моделирования путей коммуникации между целями развертывания используется ассоциация.

Путь коммуникации (communication path) является ассоциацией между двумя целями развертывания, посредством которой они обладают способностью обмениваться сигналами и сообщениями.

Путь коммуникации является ассоциацией, которая может быть определена только между целями развертывания для моделирования передачи сигналов и сообщений между ними. В языке UML 2.0 путь коммуникации является подклассом ассоциации. Концами пути коммуникации как ассоциации являются типизированные цели развертывания. Пути коммуникации между экземплярами узлов указываются посредством экземпляра ассоциации — связи.

Графически пути коммуникации изображаются в форме обычной линии ассоциации или связи. При этом концы путей коммуникации могут иметь кратности, которые определяются согласно общему определению ассоциации. В качестве примера можно привести фрагмент диаграммы развертывания для

некоторой типовой архитектуры распределенной базы данных, которая состоит из узлов с именами `СерверБазДанных`, `СерверПриложений`, `КорпоративнаяСеть` и `РабочаяСтанция`, соединенными между собой путями коммуникации (рис. 12.10).

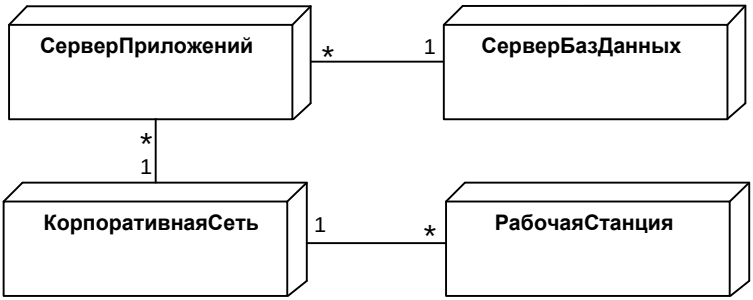


Рис. 12.10. Пути коммуникации между типами узлов

12.5. Стереотипы узлов

На диаграммах развертывания могут быть использованы специальные условные обозначения для различных физических устройств, графическое изображение которых проясняет назначение или выполняемые устройством функции. Однако пользоваться этой возможностью разработчикам следует с осторожностью, помня о том, что одно из основных достоинств языка UML следует из его названия — унификация графических изображений элементов модели.

Наиболее известными являются два графических стереотипа узла, которые используются в CASE-средстве IBM Rational Rose. Первый из них обозначает *ресурсоемкий узел* (processor), под которым понимается узел с наличием процессора и памяти, необходимых для выполнения исполняемых артефактов. Он изображается в форме куба с боковыми гранями, окрашенными в серый цвет (рис. 12.11, *а*). Второй стереотип в форме обычного куба обозначает *устройство* (device), под которым в среде IBM Rational Rose понимается узел без процессора и оперативной памяти (рис. 12.11, *б*). На этом типе узлов не могут размещаться исполняемые артефакты программной системы, что, формально говоря, несколько отличается от определения устройства в языке UML 2.0.

Кроме рассмотренных текстовых и графических стереотипов для узлов разработчики могут предложить также собственные графические стереотипы, которые улучшают наглядность представления диаграмм развертывания. Например, рабочую станцию можно изобразить как в виде ресурсоемкого

узла (рис. 12.11, *а*), так и в форме рисунка внешнего вида компьютера. Соответственно, консоль может быть изображена в виде рисунка клавиатуры. В подобных случаях разработчик должен обладать, в дополнение к основным, еще и художественными способностями. Впрочем, последнее отнюдь не является обязательным при использовании современных компьютерных графических программ.

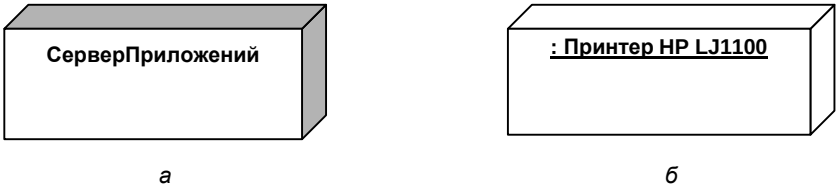


Рис. 12.11. Графические стереотипы: *а* — ресурсоемкого узла; *б* — устройства

Хотя возможность включения людей и персонала в семантику узла не рассматривается в нотации языка UML, подобное расширение понятие узла позволяет создавать в нотации UML модели самых различных систем, включая бизнес-процессы и технические комплексы. Действительно, для разработки моделей бизнес-процессов компании может оказаться удобным рассматривать в качестве узлов ресурсы системы, в частности — организационные подразделения или должности отдельных сотрудников.

Все эти вопросы являются предметом рассмотрения дополнительных профилей, написать один из которых можно предложить читателям в качестве упражнения для самостоятельной работы.

* * *

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм развертывания (табл. 12.1).

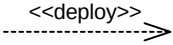
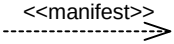
Таблица 12.1. *Графические узлы и пути на диаграммах развертывания*

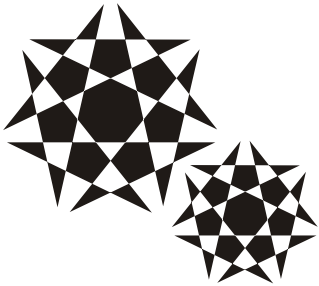
Тип графического элемента	Нотация	Ссылка на раздел
Узел (node)	Имя узла	Узел *(см. разд. 12.1)*

***Таблица 12.1** (продолжение)*

Тип графического элемента	**Нотация**	**Ссылка на раздел**
Артефакт (artifact)	<<artifact>> **Имя артефакта**	Артефакт *(см. разд. 12.2)*
Узел с размещенным артефактом	**Имя узла** <<artifact>> **Имя артефакта**	Развертывание *(см. разд. 12.4.1)*
Экземпляр узла (instance node) с размещенным экземпляром артефакта (instance artifact)	**: Имя узла** <<artifact>> **: Имя артефакта**	Развертывание *(см. разд. 12.4.1)*
Экземпляр узла с размещенными экземплярами артефактов	<<executionEnvironment>> **: Имя узла** **Артефакт 1** **Артефакт 2** **Артефакт 3**	Развертывание *(см. разд. 12.4.1)*
Спецификация развертывания (deployment specification)	<<deployment spec>> **Имя спецификации**	Спецификация развертывания *(см. разд. 12.3)*
Спецификация развертывания со свойствами	<<deployment spec>> **Имя спецификации** свойство: Тип свойства	Спецификация развертывания *(см. разд. 12.3)*
Спецификация развертывания со значениями свойств	<<deployment spec>> **: Имя спецификации** свойство: значение	Спецификация развертывания *(см. разд. 12.3)*
Артефакт с аннотируемыми свойствами развертывания	<<artifact>> **Имя артефакта** {свойство=значение}	Спецификация развертывания *(см. разд. 12.3)*

Таблица 12.1 *(окончание)*

Тип графического элемента	Нотация	Ссылка на раздел
Путь коммуникации (communication path)		Путь коммуникации *(см. разд. 12.4.3)*
Зависимость (dependency)		Отношения на диаграмме развертывания *(см. разд. 12.4)*
Развертывание (deployment)		Развертывание *(см. разд. 12.4.1)*
Манифестация (manifestation)		Манифестация *(см. разд. 12.4.2)*

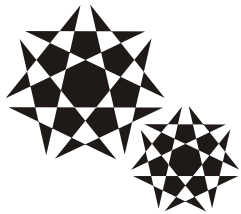


Часть III

Анализ и проектирование с использованием нотации UML 2.0 и CASE-средства Borland® Together® Designer 2005

Заключительная, третья, часть книги посвящена рассмотрению особенностей реализации языка UML 2.0 в Borland® Together® Designer 2005. Это средство было одним из первых на рынке CASE-средств, в котором появилась поддержка нотации UML 2.0. Возможно, для профессиональных программистов наибольший интерес представляет тот факт, что Borland® Together® Designer 2005 входит в состав интегрированной среды программирования Borland® Delphi® Studio 2006 в качестве встроенного модуля. Это дает возможность разработки специальных проектов с поддержкой моделей в нотации языка UML 1.*x*/2.0 с автоматической генерацией и синхронизацией программного кода на всех языках среды. В этой части книги описываются все основные возможности программы Borland® Together® Designer 2005 в контексте использования базовых конструкций языка UML 2.0 в процессе разработки приложений, управляемых моделями.

Глава 13



Особенности реализации графической нотации языка UML 2.0 в среде Borland® Together® Designer 2005

Появление на рынке программных продуктов первых CASE-средств (Computer Aided Software Engineering) ознаменовало новый этап развития программной инженерии, характерными особенностями которого являются существенное сокращение сроков реализации программных проектов, поддержка методологии групповой разработки и ориентация на визуальные средства моделирования и документирования компонентов программного обеспечения.

Первоначальной областью применения CASE-средств были приложения корпоративных баз данных, которые требовали серьезных усилий при разработке своих концептуальных схем. Автоматическая генерация программного кода на основе предварительно разработанной концептуальной схемы оказалась настолько конструктивной, что стимулировала появление CASE-средств в составе коммерческих баз данных различных фирм.

Начальный этап развития CASE-технологий характеризовался тем, что разные фирмы предлагали собственные средства визуального представления концептуальных схем. Зачастую выбор разработчиками того или иного CASE-средства определялся простотой нотации поддерживаемого средством языка представления схем и диаграмм. Появление первых стандартов в этой области лишь на короткое время стабилизировало ситуацию. Конкуренция среди фирм-производителей программного обеспечения потребовала от CASE-средств реализации объектно-ориентированной технологии разработки программ и поддержки широкого диапазона языков программирования.

В настоящее время рынок CASE-средств насчитывает десятки инструментов, которые предназначены для моделирования и разработки диаграмм в нотации

языка UML. Среди них особое место занимают средства, которые поддерживают нотацию последней версии языка UML 2.0. Одним из первых таких средств, которое оказалось доступным для разработчиков, стала программа Borland® Together® Designer 2005. Именно по этой причине данное средство выбрано в качестве базового для иллюстрации возможностей инструментальной поддержки языка UML 2.0 и процесса разработки визуальных моделей в соответствующей нотации.

Среди причин, сдерживающих применение CASE-средств и определяющих контраст их популярности среди западных и отечественных разработчиков программ, следует отметить, в первую очередь, масштабность проектов и различие в технологиях создания программ. С одной стороны, необходимость автоматизации анализа и проектирования программных систем на базе CASE-технологии начинает осознаваться только тогда, когда выполняемый проект является достаточно сложным и масштабным. В противном случае для написания программ вполне достаточно обычных инструментов разработчика.

С другой стороны, реализация масштабных проектов под силу группе разработчиков высокой квалификации, в которую, кроме программистов, должны входить системные аналитики, бизнес-аналитики, системотехники, тестировщики и специалисты других категорий. При этом непременным условием успешной работы является соответствующая квалификация специалистов, в том числе и знание ими базовой нотации языка UML 2.0, а обеспечение групповой работы над проектом требует дополнительных средств для обеспечения совместимости его составных частей.

13.1. Общая характеристика CASE-средства Borland® Together® Designer 2005

CASE-средство Borland® Together® Designer 2005 (далее — программа Together Designer) разработано компанией TogetherSoft Corporation, которая является филиалом известной корпорации Borland Software. Со времени своего появления программа Together Designer претерпела серьезную эволюцию и в настоящее время представляет собой современный интегрированный инструментарий для моделирования и разработки программных систем.

Программы семейства Together различаются между собой диапазоном реализованных в них возможностей. Кроме программы Together Designer в состав данного семейства входят средства Borland Together Designer Community Edition (сокращенно — CE) и Borland Together Developer. При этом нотация

языка UML 2.0 поддерживается только в программах Together Designer и Borland Together CE. Графический редактор Together Designer вошел в состав интегрированной среды программирования Borland® Delphi® Studio 2006. По этой причине программа Together Designer была выбрана в качестве базового CASE-средства для иллюстрации особенностей разработки канонических диаграмм языка UML 2.0.

Программа Together Designer обладает следующими возможностями:

- разработка проектов графических моделей программных приложений в нотации языка UML версии 1.4;
- преобразование проектов графических моделей, разработанных в нотации языка UML версии 1.4, в нотацию языка UML версии 2.0;
- разработка логических диаграмм данных типа "сущность — связь" в нотации ER;
- генерация программного кода на языке Java на основе разработанной диаграммы классов в нотации языка UML 1.4/2.0;
- поддержка языка объектных ограничений OCL для формальной спецификации ограничений различных элементов моделей;
- возможность использования *паттернов проектирования* (design patterns) в разрабатываемых моделях программных приложений;
- экспорт и импорт моделей в формате XMI 1.2;
- импорт моделей, разработанных в среде IBM Rational Rose 2003 Enterprise Edition;
- интеграция со специальными программами управления требованиями: Borland CaliberRM и IBM RequisitePro;
- интеграция со специальными программами контроля версий: Borland StarTeam, IBM ClearCase, MS Visual Source Safe и CVS.

В контексте разработки графических моделей принципиальное значение имеет то обстоятельство, что программа Together Designer обладает возможностью разработки двух типов проектов: как в нотации языка UML 1.4, так и в нотации языка UML 2.0. При этом из 13 канонических диаграмм нотации языка UML 2.0 программа Together Designer поддерживает следующие девять диаграмм:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграмма композитной структуры;
- диаграмма последовательности;

- ❒ диаграмма деятельности;
- ❒ диаграмма конечного автомата;
- ❒ диаграмма коммуникации;
- ❒ диаграмма компонентов;
- ❒ диаграмма развертывания.

Цель знакомства с этим инструментарием — осветить основные особенности реализации языка UML 2.0 на уровне разработки отдельных диаграмм. Поэтому далее описываются лишь основные правила и рекомендации, необходимые для разработки визуальных моделей в форме канонических диаграмм языка UML 2.0, реализованные в среде Together Designer.

13.2. Особенности рабочего интерфейса Borland Together Designer 2005

В CASE-средстве Borland Together Designer 2005 реализованы общепринятые стандарты на рабочий интерфейс программы, аналогично известным средам визуального программирования. При первом запуске программы появляется диалоговое окно с предложением ввести лицензионный номер программы либо имя и расположение файла активизации программы. Можно либо ввести этот номер, либо отказаться от ввода. Однако указание имя файла активизации является обязательным, поскольку только в этом случае произойдет загрузка программы Together Designer.

После загрузки программы в среде Microsoft® Windows® 2000/XP на экране появится соответствующий рабочий интерфейс (рис. 13.1).

Рабочий интерфейс программы Together Designer состоит из различных элементов, основными из которых являются:

- ❒ главное меню;
- ❒ стандартная панель инструментов;
- ❒ окно проекта;
- ❒ окно навигатора модели;
- ❒ окно навигатора диаграмм;
- ❒ окно инспектора;
- ❒ рабочая область изображения диаграммы или окно диаграммы;
- ❒ стандартная панель инструментов диаграммы;
- ❒ специальная панель инструментов диаграммы;
- ❒ окно истории.

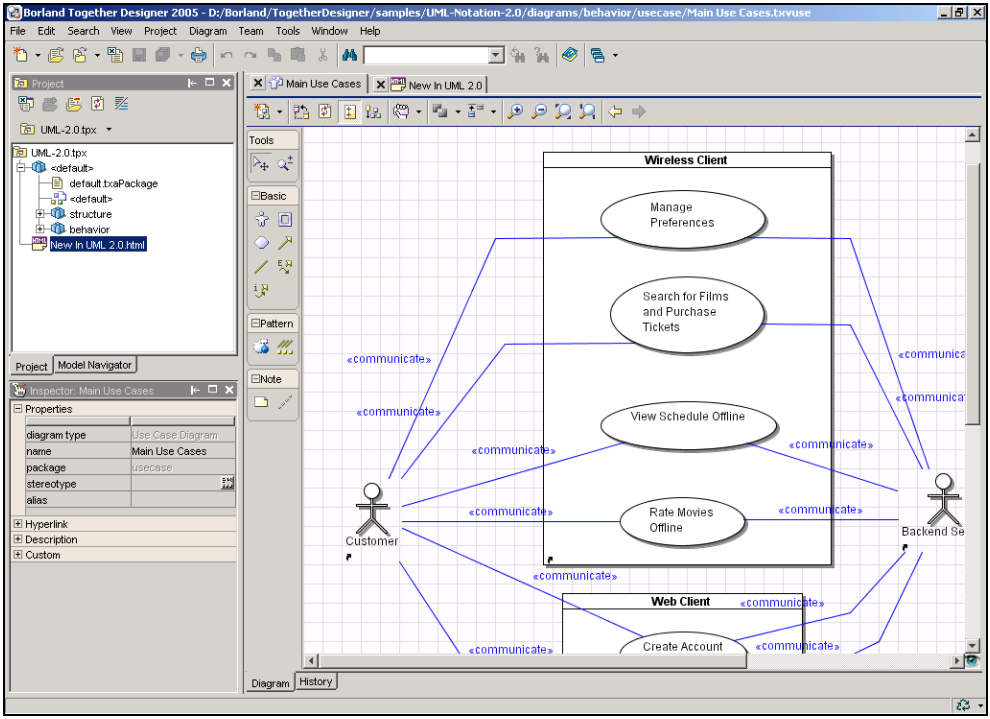


Рис. 13.1. Общий вид рабочего интерфейса программы Together Designer

Отдельные окна сгруппированы вместе, чтобы занимать меньше места на экране. В этом случае доступ к ним осуществляется переключением между соответствующими вкладками. Некоторые окна по умолчанию могут отсутствовать на экране, однако они могут быть визуализированы с помощью операции главного меню: **View | Panes** (Вид | Панели). Далее кратко рассматривается назначение и основные функции каждого из элементов рабочего интерфейса.

13.2.1. Главное меню

Главное меню средства Together Designer выполнено в общепринятом стандарте и выглядит так, как показано на рис. 13.2.

Рис. 13.2. Внешний вид главного меню программы

Отдельные пункты меню объединяют сходные операции, относящиеся ко всему проекту в целом. Некоторые из пунктов меню содержат хорошо знакомые функции, такие как открытие проекта, вывод на печать диаграмм, копирование в буфер и вставка из буфера различных элементов диаграмм. Другие настолько специфичны, что могут потребовать дополнительных усилий на изучение. К таким операциям относятся, например, операции интеграции со специальными программами управления требованиями или контроля версий. Более подробно операции главного меню рассматриваются далее в этой главе.

13.2.2. Стандартная панель инструментов

Стандартная панель инструментов располагается ниже строки главного меню и выглядит так, как показано на рис. 13.3. Некоторые из инструментов могут быть недоступными для разрабатываемого проекта, пока не будут выполнены некоторые специальные действия. Стандартная панель инструментов обеспечивает быстрый доступ к тем командам меню, которые выполняются разработчиками наиболее часто.

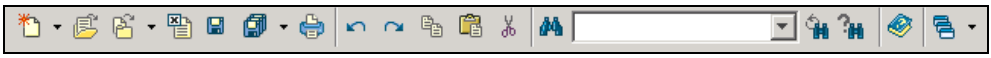


Рис. 13.3. Внешний вид стандартной панели инструментов

Пользователь может настроить внешний вид этой панели по своему усмотрению. Для этого следует выполнить операцию главного меню **View | Toolbars** (Вид | Панели инструментов) и выбрать те группы кнопок, которые нужно показать или скрыть на стандартной панели инструментов. Назначение отдельных кнопок стандартной панели инструментов приводится в табл. 13.1.

***Таблица 13.1.** Назначение кнопок стандартной панели инструментов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	New	Вызывает диалоговое окно с галереей объектов для создания нового проекта или новой диаграммы в существующем проекте
	Open File	Вызывает стандартное диалоговое окно открытия внешнего файла с диска
	Reopen	Открывает список файлов, с которыми пользователь работал в последнее время

Таблица 13.1 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Close	Закрывает активное окно с диаграммой или файлом в рабочей области изображения диаграммы
	Save	Сохраняет активную диаграмму во внешнем файле модели
	Save All	Сохраняет все внесенные в разрабатываемую модель изменения во внешнем файле с расширением TPX
	Print	Вызывает диалоговое окно настройки для печати разрабатываемой диаграммы на подключенном к компьютеру принтере
	Undo	Отменяет последовательно выполненные действия по редактированию элементов модели
	Redo	Восстанавливает свойства элементов диаграммы после отмены выполнения последнего действия по редактированию
	Copy	Копирует выделенные элементы диаграммы и их свойства в буфер обмена
	Paste	Вставляет элементы диаграммы и их свойства из буфера обмена в текущую диаграмму
	Cut	Вырезает выделенные элементы диаграммы и их свойства и помещает их в буфер обмена
	Find	Вызывает диалоговое окно поиска текста в текущей диаграмме или во всех диаграммах проекта
	Find Text	Поле ввода для поиска текста во всех диаграммах модели
	Search Again	Позволяет перейти к следующему найденному элементу после выполнения поиска
	Replace	Вызывает диалоговое окно поиска и замены текста в текущей диаграмме или во всех диаграммах проекта
	Help Topic	Вызывает программу просмотра справочной системы, открытую на вкладке **Содержание**
	Workspaces	Содержит список операций с рабочими пространствами проекта

13.2.3. Окно проекта

Окно проекта по умолчанию располагается в левой части рабочего интерфейса ниже стандартной панели инструментов и имеет вид, представленный на рис. 13.4.

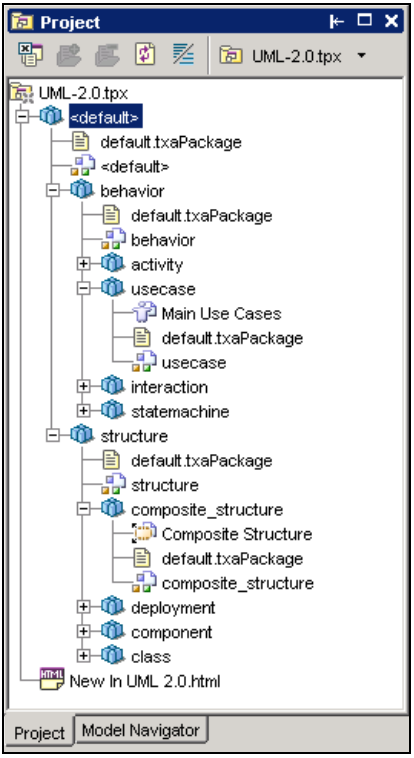


Рис. 13.4. Окно проекта с иерархическим представлением структуры проекта

В окне проекта отображается представление файлов проекта в виде иерархической структуры, которая содержит все диаграммы модели и соответствующие им файлы. С помощью данного окна можно быстро визуализировать любую диаграмму модели проекта или просмотреть содержание соответствующего текстового файла формата XML. Любая диаграмма, которую разработчик добавляет в модель, сразу отображается в окне проекта. Чтобы визуализировать нужную диаграмму в окне диаграммы, следует выполнить двойной щелчок левой кнопкой мыши на имени этой диаграммы в окне проекта.

Размеры окна проекта можно изменить. Для этого нужно переместить мышью границу внешней рамки окна, либо развернуть данное окно на весь

экран, для чего следует воспользоваться средней кнопкой в правом верхнем углу данного окна. Окно проекта при необходимости можно скрыть, воспользовавшись для этого правой кнопкой в правом верхнем углу данного окна. Отсутствующее на экране окно проекта может быть визуализировано с помощью операции главного меню **View | Panes | Project** (Вид | Панели | Проект) или комбинации клавиш <Ctrl>+<Alt>+<P>. Наконец, можно свернуть это окно и поместить его пиктограмму в левую область рамки рабочего интерфейса, воспользовавшись для этого левой кнопкой в правом верхнем углу данного окна.

13.2.4. Окно навигатора модели

Окно навигатора модели по умолчанию располагается в левой части рабочего интерфейса под окном проекта в форме отдельной вкладки (рис. 13.5).

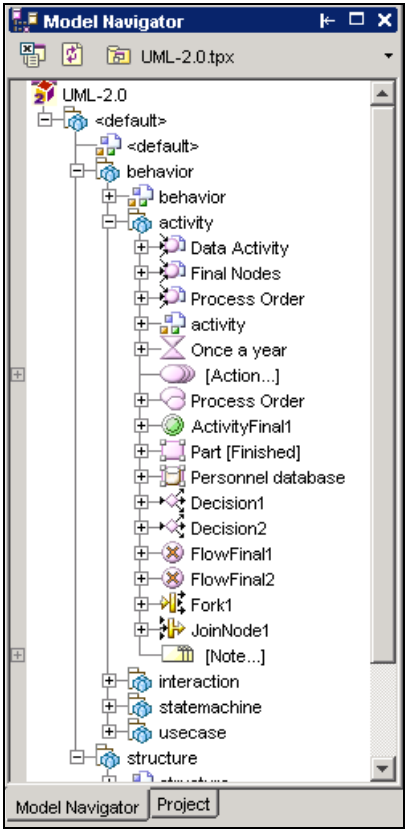


Рис. 13.5. Окно навигатора модели с иерархическим представлением ее структуры

В окне навигатора модели все элементы отображаются в виде иерархической структуры, которая упрощает навигацию и позволяет отыскать любой элемент модели в разрабатываемом проекте. При этом каждый новый элемент, который разработчик добавляет в модель, сразу отображается в окне навигатора модели. Чтобы визуализировать нужный элемент в окне диаграммы, следует выполнить двойной щелчок левой кнопкой мыши на имени этого элемента в окне навигатора модели. При этом откроется диаграмма, которая содержит выбранный элемент, и окно инспектора со свойствами этого элемента.

Отсутствующее на экране окно навигатора модели может быть визуализировано с помощью операции главного меню **View | Panes | Model Navigator** (Вид | Панели | Навигатор модели). Размеры окна навигатора модели также можно изменить, развернуть его на весь экран, скрыть, свернуть и поместить его пиктограмму в левую область рамки рабочего интерфейса. Для этого следует выполнить действия, аналогичные операциям, рассмотренным в *разд. 13.2.3* для окна проекта.

13.2.5. Окно навигатора диаграмм

Окно навигатора диаграмм может отсутствовать на экране, а после отображения окно навигатора диаграмм появится в виде отдельной вкладки в окне проекта (рис. 13.6).

В окне навигатора диаграмм все диаграммы отображаются в виде иерархической структуры, которая упрощает навигацию и позволяет отыскать любую диаграмму и ее элемент в разрабатываемом проекте. Любая диаграмма и элемент, которые разработчик добавляет в модель, сразу отображаются в окне навигатора диаграмм. Чтобы визуализировать нужный элемент или диаграмму в окне диаграммы, следует выполнить двойной щелчок левой кнопкой мыши на имени этого элемента или диаграммы в окне навигатора диаграмм. При этом откроется диаграмма, в которой содержится выбранный элемент, и окно инспектора со свойствами этого элемента.

Отсутствующее на экране окно навигатора модели может быть визуализировано с помощью операции главного меню: **View | Panes | Diagrams Navigator** (Вид | Панели | Навигатор диаграмм). Размеры окна навигатора модели также можно изменить, развернуть его на весь экран, скрыть, свернуть и поместить его пиктограмму в левую область рамки рабочего интерфейса. Для этого следует выполнить действия, аналогичные операциям, рассмотренным в *разд. 13.2.3* для окна проекта.

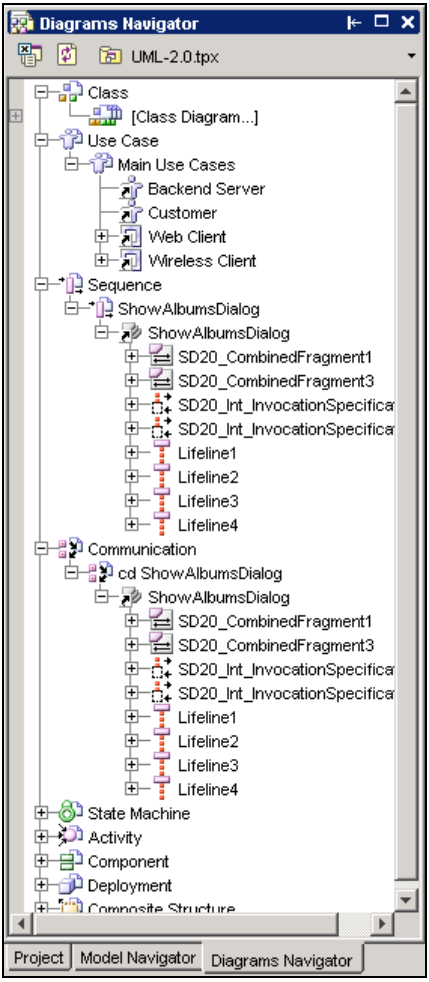


Рис. 13.6. Окно навигатора диаграмм

13.2.6. Окно инспектора

Окно инспектора по умолчанию располагается в левой части рабочего интерфейса ниже окна проекта и имеет вид, представленный на рис. 13.7.

В окне инспектора отображаются свойства и другие характеристики выбранного элемента модели, сгруппированные по категориям. Это окно предназначено для редактирования свойств одного или нескольких элементов диаграммы. В случае выбора нескольких элементов на диаграмме в окне инспектора будут доступны для редактирования только общие для них свойства. Отсутст-

вующее на экране окно инспектора может быть визуализировано с помощью операции главного меню: **View | Panes | Inspector** (Вид | Панели | Инспектор).

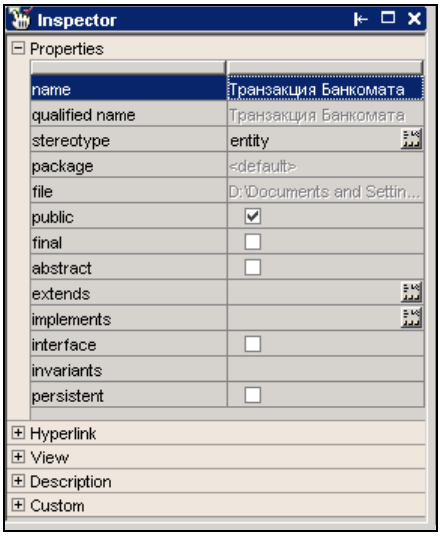


Рис. 13.7. Окно инспектора для выбранного элемента модели

13.2.7. Окно диаграммы

Окно диаграммы, также называемое окном содержания, является основной областью ее рабочего интерфейса, в которой визуализируются различные диаграммы проекта или текстовые файлы проекта в формате HTML или XML. Именно в этом окне происходит работа с редактором диаграмм. По умолчанию окно диаграммы располагается в правой части рабочего интерфейса, однако его расположение и размеры можно изменить. При разработке нового проекта, если не был использован мастер проектов, окно диаграммы представляет собой чистую область, не содержащую никаких элементов модели.

Название диаграммы, которая располагается в данном окне, указывается в строке заголовка программы или, если окно развернуто во весь экран, в строке заголовка окна диаграммы. Каждая диаграмма проекта отображается в отдельном окне (рис. 13.8), при этом активным может быть только одно из них.

Переключение между диаграммами в окне диаграммы можно осуществить с помощью соответствующей вкладки, которая содержит имя открытой диаграммы. При открытии отдельной диаграммы проекта изменяется внешний вид специальной панели инструментов, которая настраивается под конкретный

тип канонической диаграммы. Отсутствующее на экране окно диаграммы модели может быть визуализировано с помощью операции главного меню **View | Panes | Content** (Вид | Панели | Содержимое) или комбинации клавиш <Ctrl>+<Alt>+<C>. Окно диаграммы модели может быть развернуто на весь экран с помощью комбинации клавиш <Ctrl>+<Alt>+<Z>.

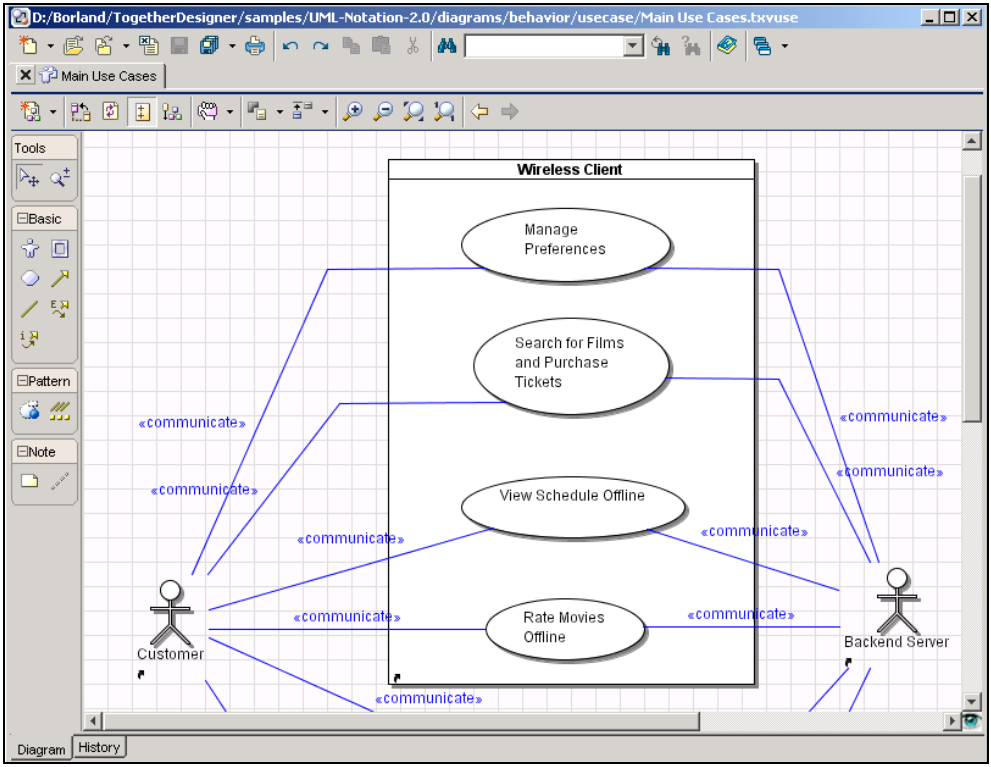


Рис. 13.8. Внешний вид окна диаграммы модели

13.2.8. Стандартная панель инструментов окна диаграммы модели

Стандартная панель инструментов окна диаграммы модели располагается в верхней части окна диаграммы и выглядит так, как показано на рис. 13.9. Некоторые из кнопок могут быть недоступными для редактируемой диаграммы, пока не будут выполнены определенные действия. Стандартная панель инструментов обеспечивает быстрый доступ к тем командам меню, которые выполняются разработчиками наиболее часто.

Рис. 13.9. Внешний вид стандартной панели инструментов окна диаграммы модели

Назначение отдельных кнопок стандартной панели инструментов окна диаграммы модели приводится в табл. 13.2.

Таблица 13.2. *Назначение кнопок стандартной панели инструментов окна диаграммы*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	New diagram	Открывает диалоговое окно создания новой диаграммы, в котором можно выбрать тип новой диаграммы, ее имя, местоположение и описание
	Open parent diagram	Открывает родительскую диаграмму для текущей активной диаграммы. Если активная диаграмма не имеет родительской, то эта кнопка недоступна
	Update	Сохраняет все внесенные изменения в текущей версии проекта при контроле версий
	Make Compartments Expandable	Раскрывает внутренние секции всех элементов активной диаграммы. Ели кнопка отжата, то имена всех внутренних секций скрыты
	Properties	Открывает диалоговое окно свойств активной диаграммы, в котором можно настроить графическое представление элементов активной диаграммы
	Profiles	Позволяет выбрать дополнительный профиль для представления элементов активной диаграммы
	Layout	Позволяет выбрать вариант расположения элементов на активной диаграмме
	Align	Позволяет выбрать вариант выравнивания выбранных элементов на активной диаграмме
	Zoom In	Увеличивает масштаб изображения элементов активной диаграммы
	Zoom Out	Уменьшает масштаб изображения элементов активной диаграммы
	Fit In Window	Изменяет масштаб изображения элементов активной диаграммы для ее размещения полностью в окне диаграммы

Таблица 13.2 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Actual size	Возвращает масштаб изображения элементов активной диаграммы в первоначальное положение
	Back	Позволяет перейти к предыдущему элементу активной диаграммы
	Forward	Позволяет перейти к последующему элементу активной диаграммы

13.2.9. Специальная панель инструментов диаграммы модели

Специальная панель инструментов располагается между окном проекта и окном диаграммы в средней части рабочего интерфейса. Состав кнопок данной панели инструментов зависит от типа диаграммы, которая открыта и является активной в окне диаграммы. Например, для диаграммы вариантов использования специальная панель инструментов выглядит так, как показано на рис. 13.10.

Рис. 13.10. Внешний вид специальной панели инструментов для диаграммы вариантов использования

Назначения кнопок специальной панели инструментов можно узнать из всплывающих подсказок, появляющихся после задержки указателя мыши

над соответствующей кнопкой. Более подробно состав и назначение отдельных кнопок специальных панелей инструментов будут рассмотрены далее при построении соответствующих типов канонических диаграмм.

13.2.10. Окно истории

Окно истории присутствует в форме отдельной вкладки в нижней части окна диаграммы (рис. 13.11). Переключиться в это окно можно, нажав на соответствующую вкладку **History** (История).

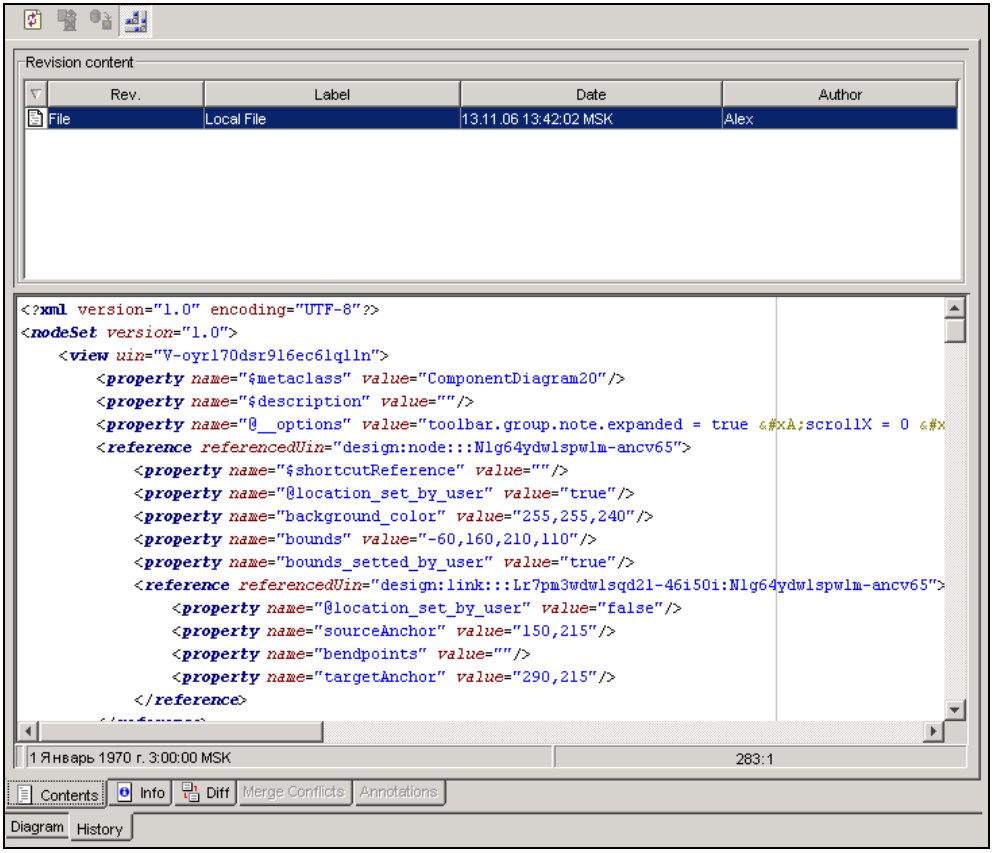


Рис. 13.11. Внешний вид окна истории

Окно истории предназначено для просмотра текстового файла формата XML для диаграммы, которая открыта в окне диаграммы. Информация о любом элементе, который разработчик добавляет на диаграмму, сразу заносится

в текстовый файл диаграммы. Дополнительно в этом окне содержится информация о времени последнего просмотра или редактирования диаграммы, а также имя соответствующего пользователя. В окне истории можно также сравнить соответствующие текстовые файлы для нескольких версий диаграммы и выполнить их объединение.

13.3. Назначение операций главного меню

Операции главного меню позволяют пользователю выполнять все необходимые действия в ходе работы над проектом. Возможности программы Together Designer во многом определяются составом операций главного меню, которые рассматриваются далее в этой главе.

13.3.1. Пункт меню *File* (Файл)

Пункт меню **File** (Файл) главного меню содержит следующие операции (рис. 13.12):

- **New** — вызывает диалоговое окно с галереей объектов для создания нового проекта или новой диаграммы в существующем проекте;
- **New Diagram** — вызывает диалоговое окно создания новой диаграммы, в котором можно выбрать тип новой диаграммы, ее имя, местоположение и описание;
- **New Project** — создает новый проект Together Designer с именем по умолчанию untitled1;
- **Open Project** — вызывает стандартное диалоговое окно открытия файла проекта с расширением TPX;
- **Open File** — вызывает стандартное диалоговое окно открытия текстового файла в окне содержимого;
- **Reopen** — открывает список файлов и диаграмм, с которыми пользователь работал в последнее время;
- **Close Projects** — вызывает диалоговое окно со списком всех открытых проектов для их закрытия;
- **Close <*активная диаграмма*>** — закрывает окно с активной диаграммой;
- **Close All Except <*активная диаграмма*>** — закрывает все открытые диаграммы, за исключением текущей активной диаграммы;
- **Close** — вызывает диалоговое окно со списком всех открытых диаграмм для их закрытия;

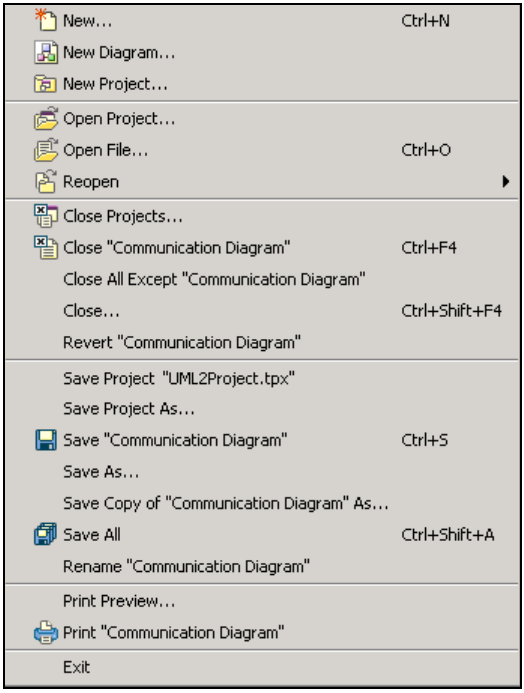


Рис. 13.12. Операции пункта меню **File** главного меню

- **Revert** <***активная диаграмма***> — позволяет вернуться к предыдущей версии активной диаграммы, помещенной под контроль версий;
- **Save Project** <***активный проект***> — позволяет сохранить разрабатываемый проект во внешнем файле на диске;
- **Save Project As** — позволяет сохранить разрабатываемый проект под другим именем во внешнем файле на диске. При этом вызывается стандартное диалоговое окно сохранения файла на диске с предложением задать имя соответствующего файла проекта;
- **Save** <***активная диаграмма***> — позволяет сохранить изменения в активной диаграмме проекта;
- **Save As** — позволяет сохранить активную диаграмму проекта под другим именем;
- **Save Copy of** <***активная диаграмма***> — позволяет сохранить копию активной диаграммы проекта под другим именем;
- **Save All** — позволяет сохранить внесенные изменения во всех диаграммах проекта;

- **Rename** <***активная диаграмма***> — позволяет изменить имя активной диаграммы проекта;
- **Print Preview** — позволяет просмотреть в отдельном окне макет страницы с активной диаграммой для последующей ее распечатки на принтере;
- **Print** <***активная диаграмма***> — вызывается стандартное диалоговое окно настройки свойств печати активной диаграммы;
- **Exit** — закрывает программу Together Designer.

13.3.2. Пункт меню *Edit* (Редактирование)

Пункт меню **Edit** (Редактирование) содержит следующие операции (рис. 13.13):

- **Undo** — отменяет последовательно выполненные действия по редактированию элементов модели;
- **Redo** — восстанавливает свойства элементов диаграммы после отмены выполнения последнего действия по редактированию;
- **Cut** — вырезает выделенные элементы диаграммы и их свойства и помещает их в буфер обмена;
- **Copy** — копирует выделенные элементы диаграммы и их свойства в буфер обмена;
- **Paste** — вставляет элементы диаграммы и их свойства из буфера обмена в текущую диаграмму;
- **Clone** — позволяет создавать новый элемент диаграммы как копию выбранного элемента для его дальнейшего клонирования;
- **Delete** — вызывает диалоговое окно для выбора варианта удаления выбранных элементов из разрабатываемого проекта (выбор — да) или только из текущей диаграммы (выбор — нет);

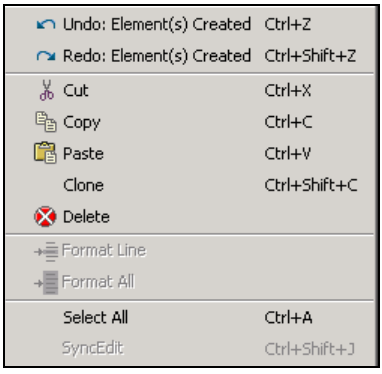


Рис. 13.13. Операции пункта меню **Edit** главного меню

- **Format Line** — форматирует выбранную строку текста;
- **Format All** — форматирует все строки текста;
- **Select All** — выделяет все элементы на активной диаграмме разрабатываемого проекта;
- **SyncEdit** — выполняет синхронное редактирование нескольких элементов.

13.3.3. Пункт меню *Search* (Поиск)

Пункт меню **Search** (Поиск) позволяет отображать на экране различные элементы рабочего интерфейса и изменять графическое представление диаграмм, содержит следующие операции (рис. 13.14):

- **Find** — вызывает диалоговое окно поиска текста в текущей диаграмме или во всех диаграммах проекта;
- **Search Again** — позволяет перейти к следующему найденному элементу после выполнения поиска;
- **Query-Based Search** — вызывает диалоговое окно спецификации специального запроса для поиска элемента в проекте.

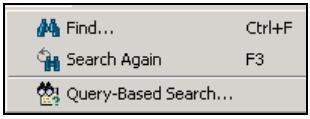


Рис. 13.14. Операции пункта меню **Search** главного меню

13.3.4. Пункт меню *View* (Вид)

Пункт меню **View** (Вид) позволяет отображать на экране различные элементы рабочего интерфейса и изменять графическое представление диаграмм, содержит следующие операции (рис. 13.15):

- **Maximize Content Pane** — позволяет развернуть окно диаграммы на весь экран;
- **Context Menu** — открывает контекстное меню для выбранного элемента проекта;
- **Window Actions** — открывает меню общих операций для выбранного элемента проекта;
- **Remove All Message Tabs** — удаляет все сообщения из окна сообщений;

- **Select Message Pane Tab** — позволяет переключаться между несколькими окнами сообщений;
- **Select Browser Splitter** — позволяет переключаться между отдельными частями окна просмотра информации;
- **Workspaces** — содержит список операций с рабочими пространствами проекта;
- **Panes** — позволяет отобразить или скрыть отдельные окна рабочего интерфейса программы;
- **Toolbars** — позволяет отобразить или скрыть отдельные группы кнопок на стандартной панели инструментов.

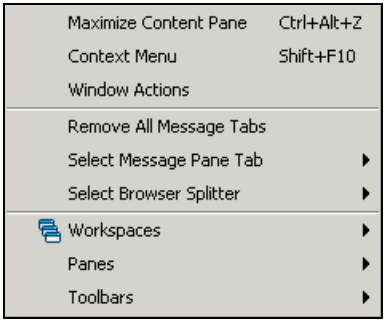


Рис. 13.15. Операции пункта меню **View** главного меню

13.3.5. Пункт меню *Project* (Проект)

Пункт меню **Project** (Проект) содержит следующие операции (рис. 13.16):

- **Add Files** — позволяет добавить файлы к проекту;
- **Refresh** — сохраняет все внесенные изменения в текущей версии проекта при контроле версий;
- **Rename <*активный проект*>** — позволяет изменить имя проекта;
- **Project Properties** — вызывает диалоговое окно настройки свойств для разрабатываемого проекта;
- **Default Project Properties** — вызывает диалоговое окно свойств по умолчанию для разрабатываемого проекта;
- **Documentation** — вызывает мастер настройки свойств для генерирования документации по проекту.

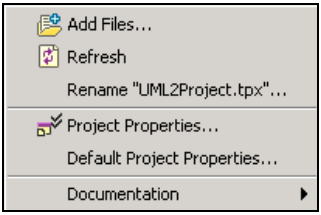


Рис. 13.16. Операции пункта меню **Project** главного меню

13.3.6. Пункт меню *Diagram* (Диаграмма)

Пункт меню **Diagram** (Диаграмма) содержит следующие операции (рис. 13.17):

- **New** — открывает диалоговое окно создания новой диаграммы, в котором можно выбрать тип новой диаграммы, ее имя, местоположение и описание;
- **Copy Image** — позволяет скопировать рисунок активной диаграммы в буфер обмена;
- **Save Image** — позволяет сохранить рисунок активной диаграммы в графическом файле одного из трех форматов;
- **Layout** — позволяет выбрать вариант расположения элементов на активной диаграмме;
- **Align** — позволяет выбрать вариант выравнивания выбранных элементов на активной диаграмме;

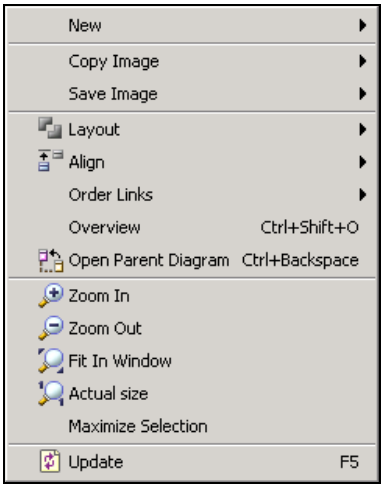


Рис. 13.17. Операции пункта меню **Diagram** главного меню

- **Order Links** — позволяет выбрать вариант выравнивания выбранных элементов и взаимосвязей между ними на активной диаграмме;
- **Overview** — отображает специальное окно обзора диаграммы;
- **Open Parent Diagram** — открывает родительскую диаграмму для текущей активной диаграммы;
- **Zoom In** — увеличивает масштаб изображения элементов активной диаграммы;
- **Zoom Out** — уменьшает масштаб изображения элементов активной диаграммы;
- **Fit In Window** — изменяет масштаб изображения элементов активной диаграммы для ее размещения полностью в окне диаграммы;
- **Actual size** — возвращает масштаб изображения элементов активной диаграммы в первоначальное положение;
- **Maximize Selection** — делает максимальным масштаб изображения выбранных элементов активной диаграммы;
- **Update** — обновляет отображение элементов активной диаграммы.

13.3.7. Пункт меню *Team* (Команда)

Состав операций этого пункта меню существенно отличается в зависимости от того, помещен или нет проект под контроль версий. Если проект помещен под контроль версий, например, с использованием программы Borland StarTeam, то пункт меню **Team** (Команда) содержит следующие операции (рис. 13.18):

- **Select Project VCS** — отображает диалоговое окно для выбора программы контроля версий;
- **Configure StarTeam** — открывает диалоговое окно настройки свойств для установления соединения с программой управления конфигурацией Borland StarTeam;
- **Update Project Group** — последовательно обновляет все проекты из группы активных проектов. Эта операция меню доступна, только если активный проект входит в группу проектов;
- **Create Patch** — позволяет создать множество изменений для активного проекта при удаленной работе над проектом;
- **Apply Patch** — позволяет применить множество изменений к активному проекту при удаленной работе над проектом;
- **File Include Lists** — отображает диалоговое окно со списком файлов, помещенных под контроль версий;

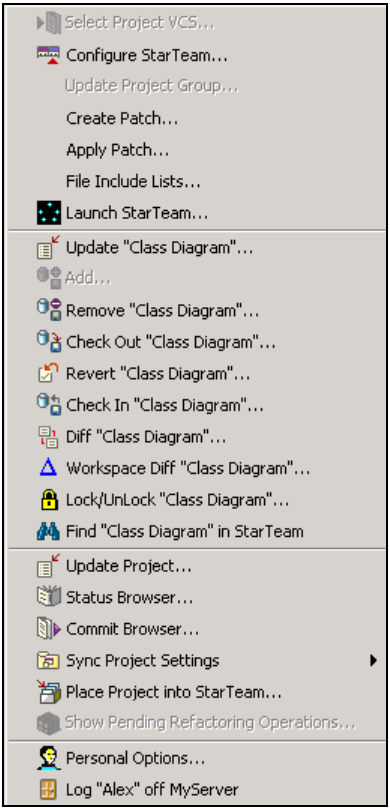


Рис. 13.18. Операции пункта меню **Team** главного меню

- **Launch StarTeam** — запускает программу управления конфигурацией StarTeam;
- **Update** <***активная диаграмма***> — позволяет сохранить все внесенные изменения в файле активной диаграммы, хранящейся в репозитории контроля версий;
- **Add** — позволяет добавить файл с активной диаграммой под контроль версий;
- **Remove** <***активная диаграмма***> — позволяет удалить файл с активной диаграммой из репозитория контроля версий;
- **Check Out** <***активная диаграмма***> — позволяет получить файл с активной диаграммой из репозитория контроля версий для редактирования;
- **Revert** <***активная диаграмма***> — позволяет отменить все изменения в редакторе диаграмм и вернуться к предыдущей версии файла активной диаграммы, помещенного под контроль версий;

- **Check In <*активная диаграмма*>** — позволяет сдать файл с активной диаграммой в репозиторий контроля версий;
- **Diff <*активная диаграмма*>** — вызывает диалоговое окно сравнения рабочей версии активной диаграммы с последней сданной версией из репозитория;
- **Workspace Diff <*активная диаграмма*>** — вызывает диалоговое окно сравнения рабочей версии активной диаграммы с полученной версией из репозитория;
- **Lock/UnLock <*активная диаграмма*>** — позволяет блокировать или разблокировать активную диаграмму при ее получении или сдаче под контроль версий;
- **Find <*активная диаграмма*> in StarTeam** — открывает панель StarTeam со списком файлов с выделенной активной диаграммой;
- **Update Project** — синхронизирует все рабочие версии файлов проекта с версиями из репозитория, объединяя все различия между ними;
- **Status Browser** — позволяет просмотреть статус всех элементов, помещенных под контроль версий, и различие между любыми двумя версиями;
- **Commit Browser** — позволяет просмотреть статус всех элементов, помещенных под контроль версий, и выполнить некоторые дополнительные операции, такие как добавление комментариев или переименование отдельных элементов;
- **Sync Project Settings** — позволяет выполнять операции по синхронизации файлов текущего проекта в рабочей области и репозитории;
- **Place Project into StarTeam** — открывает диалоговое окно мастера для размещения проекта под контроль версий программы StarTeam;
- **Show Pending Refactoring Operations** — открывает панель StarTeam в окне сообщений с информацией обо всех выполненных операциях рефакторинга, таких как переименование или перемещение элементов, которые еще не сохранены в репозитории;
- **Personal Options** — вызывает диалоговое окно с индивидуальными настройками пользователя, которые соответствуют его имени при входе в систему контроля версий;
- **Log <*имя пользователя*> off <*имя сервера*>** — позволяет отключить пользователя от сервера StarTeam с целью, например, последующего входа под другим именем.

13.3.8. Пункт меню *Tools* (Инструменты)

Пункт меню **Tools** (Инструменты) содержит следующие операции (рис. 13.19):

- **License Manager** — открывает диалоговое окно с информацией о лицензиях для всех установленных на компьютере пользователя программ компании Borland для просмотра и изменения этой информации;
- **Preferences** — открывает диалоговое окно настройки параметров проекта;
- **File Associations** — открывает диалоговое окно ассоциации с файлом проекта, который по умолчанию имеет расширение TPX;
- **Generate Java Source Code** — открывает диалоговое окно для указания папки, в которую будут помещены файлы после генерации исходного кода на языке программирования Java;
- **Export Model To XMI File** — открывает диалоговое окно для задания файла экспорта модели в формате XMI (XML-based Model Interchange);
- **Pattern** — открывает диалоговое окно для просмотра информации о паттернах или создания нового паттерна;
- **Model** — открывает диалоговое окно с целью выбора моделей для сравнения и просмотра результатов сравнения;
- **RequisitePro** — открывает дополнительное окно программы управления требованиями IBM Rational Requisite Pro с предварительно созданным в нем проектом;
- **CaliberRM** — позволяет установить соединение с программой управления требованиями Borland CaliberRM.

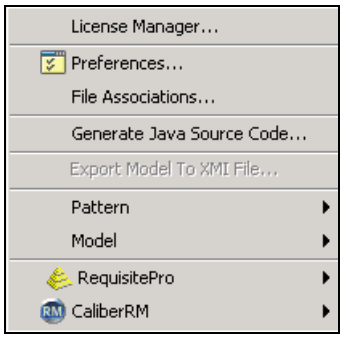


Рис. 13.19. Операции пункта меню **Tools** главного меню

13.3.9. Пункт меню *Window* (Окно)

Пункт меню **Window** (Окно) позволяет управлять окнами диаграмм и содержит следующие операции (рис. 13.20):

- секция, содержащая имена всех открытых диаграмм и файлов проекта для переключения между ними. Если открывается новая диаграмма или файл, то в этой секции появляется новая строка с именем этой диаграммы или файла, выбрав которую можно сразу переключиться в нужное окно;
- **Switch** — отображает диалоговое окно со всеми открытыми диаграммами и файлами проекта для переключения между ними;
- **Projects** — отображает все открытые проекты для переключения между ними;
- **History** — упорядочивает расположение всех открытых диаграмм;
- **Back** — позволяет перейти к предыдущему элементу активной диаграммы;
- **Forward** — позволяет перейти к последующему элементу активной диаграммы.

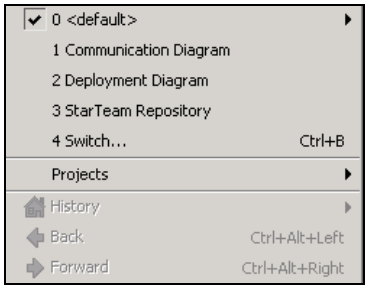


Рис. 13.20. Операции пункта меню **Window** главного меню

13.3.10. Пункт меню *Help* (Справка)

Пункт меню **Help** (Справка) содержит следующие операции (рис. 13.21):

- **Help Topics** — вызывает программу просмотра справочной системы, открытой на вкладке **Содержание**;
- **Tip of the Day** — позволяет просмотреть в отдельном окне подсказки дня;
- **About Together Designer 2005** — отображает информацию о текущей рабочей версии Together Designer.

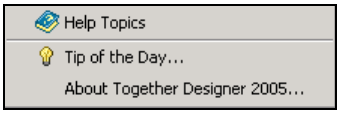
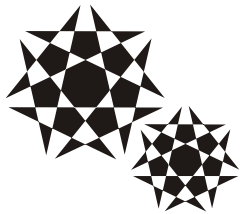


Рис. 13.21. Операции пункта меню **Help** главного меню

Глава 14



Организация работы над проектом в среде Borland Together Designer

Процесс работы над проектом состоит в последовательной разработке канонических диаграмм, которые в совокупности представляют интегрированную модель разрабатываемой программной системы. Хотя последовательность разработки диаграмм в языке UML 2.0 не определена, для этой цели можно воспользоваться рекомендациями рационального унифицированного процесса RUP. Принятый в книге порядок разработки апробирован на нескольких реальных проектах и оказался довольно удачным.

Концептуально процесс разработки канонических диаграмм заключается в размещении на диаграммах соответствующих графических элементов, установлении отношений между этими элементами, их спецификации и документировании, в соответствии с рассмотренными в *части II* правилами и нотацией языка UML 2.0. После построения модели в среде Together Designer можно сгенерировать текст программного кода на языке программирования Java. В последующем этот код следует доработать в той или иной среде программирования, например Borland® JBuilder® 2005, с целью получения исполнимых модулей программы, ориентированной на определенную вычислительную платформу.

Технически процесс разработки графических диаграмм в среде Together Designer имеет свои особенности. Разработчик выбирает необходимый графический элемент посредством нажатия соответствующей кнопки на специальной панели инструментов и размещает этот элемент на рабочем листе в окне диаграммы. После этого редактируется набор свойств этого элемента в соответствии с рассмотренной нотацией языка UML 2.0.

14.1. Разработка диаграммы вариантов использования в среде Together Designer

Работа над проектом в среде Together Designer начинается с общего анализа требований к разрабатываемой системе и построения диаграммы вариантов использования, которая отражает функциональное назначение проектируемой системы. Нотация элементов диаграммы вариантов использования и общие рекомендации по ее построению были рассмотрены в *главе 3*. В качестве разрабатываемого проекта рассматривается модель системы управления банкоматом, которая не требует дополнительного описания и обычно используется для иллюстрации элементов нотации языка UML.

14.1.1. Создание нового проекта и новой диаграммы вариантов использования

Для создания нового проекта в среде Together Designer следует выполнить операцию главного меню **File | New** (Файл | Новый). После этого появится окно с галереей объектов, внешний вид которого изображен на рис. 14.1.

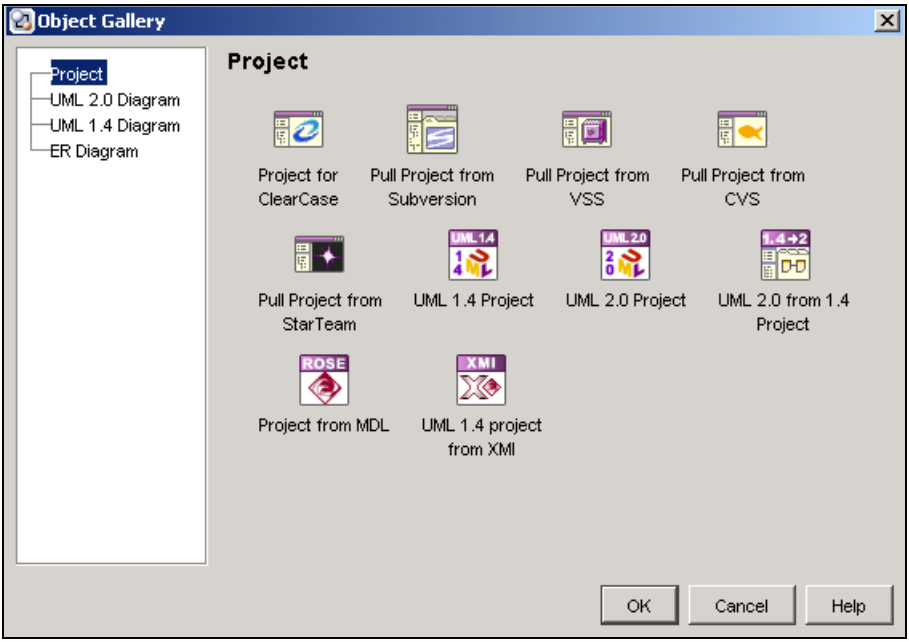


Рис. 14.1. Диалоговое окно с галереей объектов при создании нового проекта

В этом окне следует выбрать вариант проекта **UML 2.0 Project** и нажать кнопку **OK**. В результате появится первое из двух окон мастера создания проекта, внешний вид которого изображен на рис. 14.2.

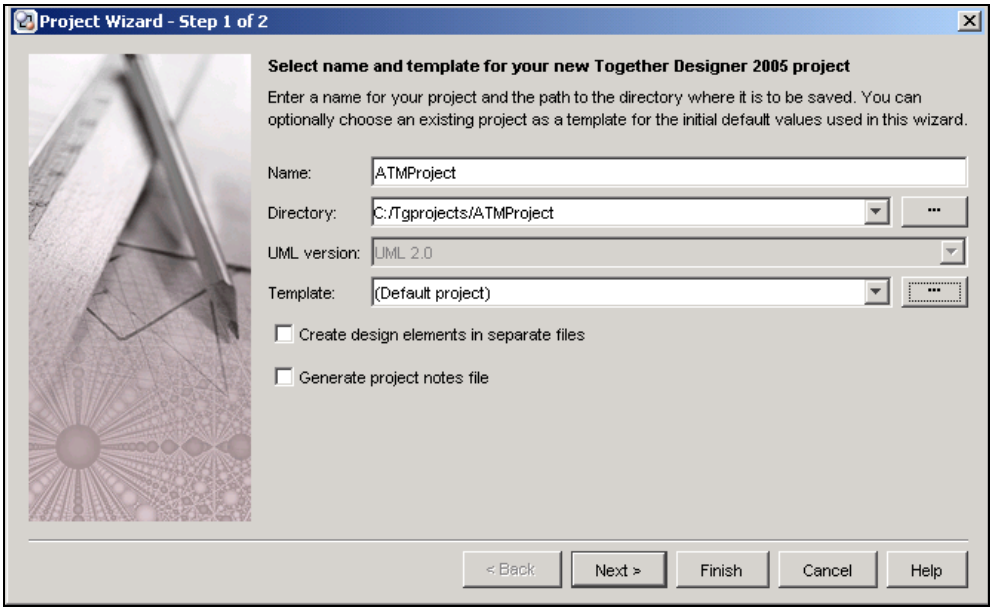


Рис. 14.2. Первое окно мастера создания нового проекта

В первой строке зададим имя нового проекта: ATMProject, а во второй — папку, в которой будут храниться рабочие файлы проекта. В качестве шаблона можно воспользоваться проектом по умолчанию, после чего следует нажать кнопку **Next** или **Finish**, если не предполагается вносить других изменений в настройки, выбранные по умолчанию.

В результате выполнения этих действий будет создан новый проект для разработки диаграмм в нотации языка UML 2.0. В окне диаграммы будет отображен чистый рабочий лист для диаграммы классов, которая выбрана программой Together Designer по умолчанию в качестве базовой, а в строке заголовка появится имя файла диаграммы классов и имя папки, в которой размещается этот файл: C:/Tgprojects/ATMProject/diagrams/default.txvClassDiagram20.

В этом проявляется важная особенность программы Together Designer: общие настройки проекта хранятся в текстовом файле ATMProject.tpx, а каждая диаграмма проекта сохраняется в отдельном текстовом файле формата XML с соответствующим расширением.

Следует заметить, что программа Together Designer позволяет использовать символы кириллицы для спецификации элементов разрабатываемых моделей. Однако при генерации текста программного кода на Java следует записывать имена и свойства классов, ассоциаций, атрибутов, операций и компонентов символами, которые поддерживаются этим языком программирования. Тем не менее в методических целях имена всех элементов модели и их свойств в рассматриваемом проекте будут записываться символами кириллицы.

Как и другие программы, Together Designer позволяет настраивать глобальные параметры среды, такие как выбор шрифта и цвета для представления различных элементов модели. Настройка шрифтов, цвета линий и графических элементов производится в специальном окне, которое вызывается с помощью операции главного меню **Tools | Preferences** (Инструменты | Параметры).

Первой диаграммой в нашем проекте будет диаграмма вариантов использования. Для разработки этой диаграммы рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами:

- с помощью операции главного меню **File | New Diagram** (Файл | Новая диаграмма);
- с помощью операции главного меню **Diagram | New | UML 2.0 Diagram** (Диаграмма | Новая | Диаграмма UML 2.0);
- с помощью операции контекстного меню в окне проекта **New | Diagram** (Новая | Диаграмма);
- нажав кнопку **New diagram** (Новая диаграмма) на стандартной панели инструментов окна диаграммы (см. табл. 13.2).

После этого появится окно с галереей объектов для выбора типа новой диаграммы, внешний вид которого изображен на рис. 14.3.

В этом окне следует выбрать тип новой диаграммы **Use Case** и нажать кнопку **OK**. В результате появится окно спецификации новой диаграммы вариантов использования, внешний вид которого изображен на рис. 14.4.

Оставим имя диаграммы без изменения, а в многострочное поле **Description** (Описание) введем текст, поясняющий назначение этой диаграммы. После этого следует нажать кнопку **OK**, если не предполагается изменять имя пакета по умолчанию. В результате выполнения этих действий будет создана и добавлена к разрабатываемому проекту новая диаграмма вариантов использования. В окне диаграммы будет отображен чистый рабочий лист для диаграммы вариантов использования (рис. 14.5).

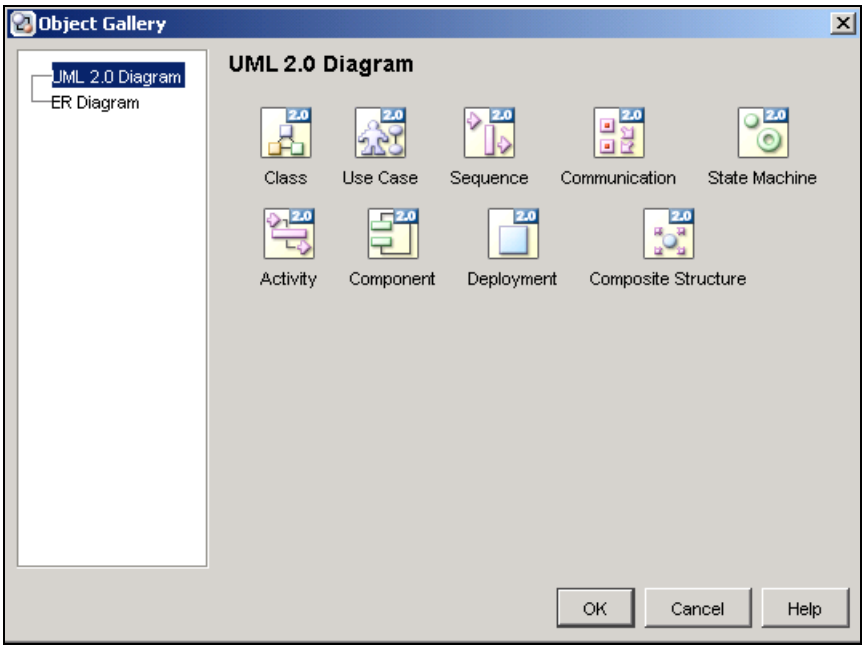


Рис. 14.3. Диалоговое окно с галереей объектов при создании новой диаграммы

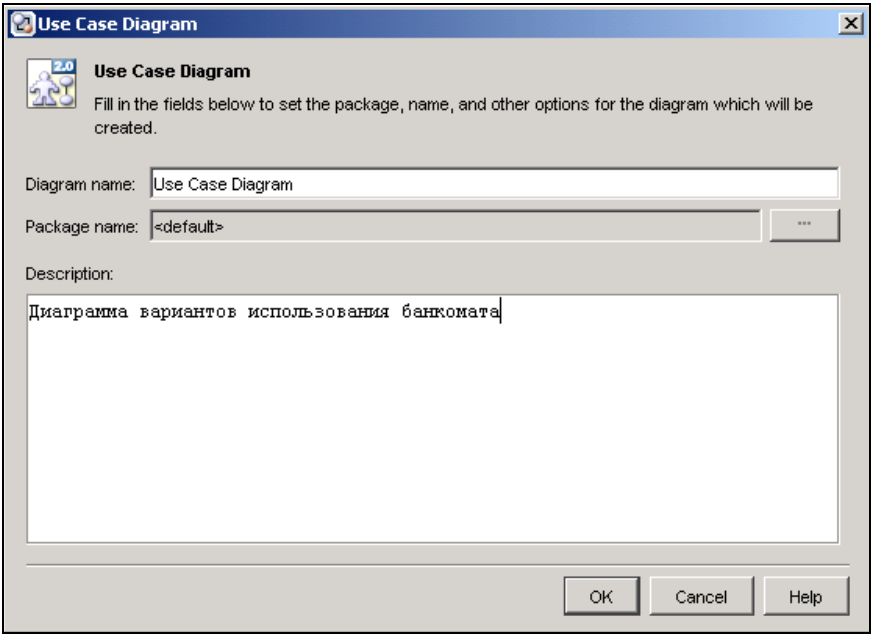


Рис. 14.4. Первое окно мастера создания нового проекта

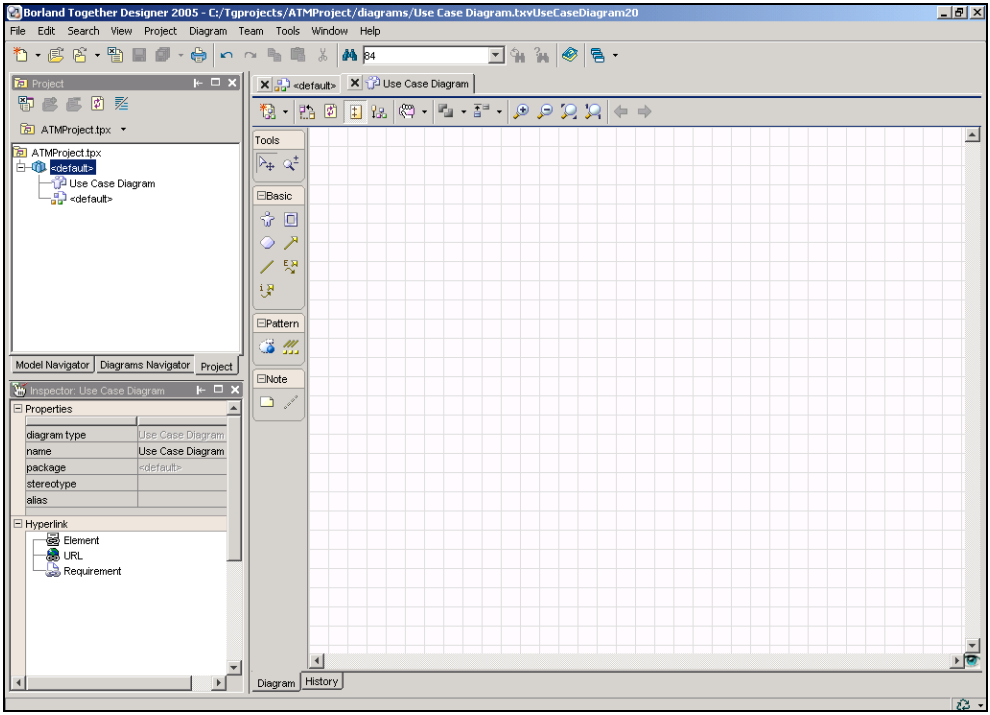


Рис. 14.5. Рабочий лист новой диаграммы вариантов использования

В строке заголовка появится имя файла диаграммы вариантов использования и имя папки, в которой размещается этот файл: C:/Tgprojects/ATMProject/ diagrams/ Use Case Diagram.txvUseCaseDiagram20. Слева от окна диаграммы появится специальная панель инструментов, которая содержит кнопки с изображением графических пиктограмм элементов, необходимых для разработки диаграммы вариантов использования (табл. 14.1). При этом назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 14.1. *Назначение кнопок специальной панели инструментов диаграммы вариантов использования*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Actor	Добавляет на диаграмму актера
	Subject	Добавляет на диаграмму границу системы

Таблица 14.1 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Use Case	Добавляет на диаграмму вариант использования
	Generalization	Добавляет на диаграмму отношение обобщения
	Association	Добавляет на диаграмму ассоциацию
	Extend	Добавляет на диаграмму отношение зависимости со стереотипом <<extend>>
	Include	Добавляет на диаграмму отношение зависимости со стереотипом <<include>>

Кроме рассмотренных кнопок специальной панели инструментов здесь также расположены кнопки с пиктограммами паттерна и связи с паттерном, а также примечания и связи примечания с аннотируемым элементом. Поскольку эти кнопки доступны для всех типов диаграмм, они в табл. 14.1 не приводятся.

14.1.2. Добавление актеров

Для добавления актера на диаграмму варианта использования нужно нажать кнопку с изображением пиктограммы актера на специальной панели инструментов и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение актера с маркерами изменения его геометрических размеров и предложенным по умолчанию именем `Actor1`, которое изменим на `Клиент Банкомата` (рис. 14.6).

Имя добавленного элемента может быть изменено разработчиком либо сразу после размещения элемента на диаграмме, либо в ходе последующей работы над проектом. Для этой цели следует выделить элемент на диаграмме и нажать клавишу <F2>. Редактирование всех свойств элемента, включая и его имя, можно выполнять в окне инспектора, которое по умолчанию располагается в левой части рабочего интерфейса ниже окна проекта (рис. 14.7).

Аналогичным образом на данную диаграмму следует добавить второго актера с именем `Банк`.

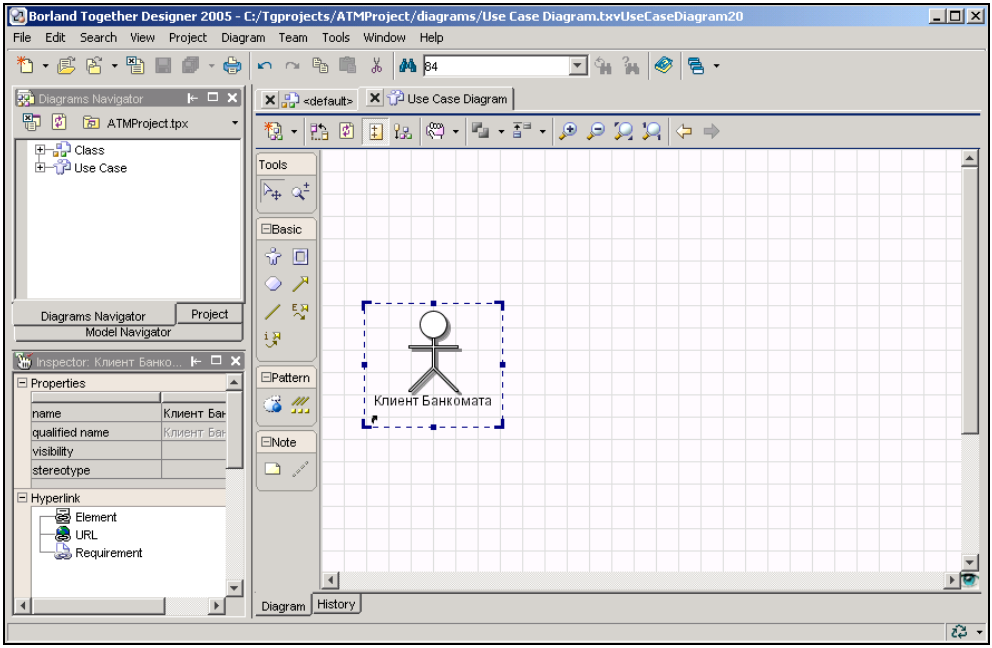


Рис. 14.6. Диаграмма вариантов использования после добавления на нее актера

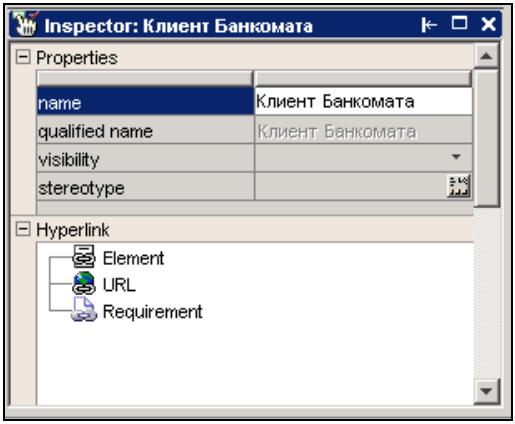


Рис. 14.7. Окно инспектора при редактировании свойств актера

14.1.3. Добавление границы системы

Для добавления на диаграмму границы системы нужно нажать на специальной панели инструментов кнопку с изображением границы системы **(Subject)** и щелкнуть левой кнопкой мыши на свободном месте диаграммы. На диаграмме

появится изображение границы системы с маркерами изменения его геометрических размеров и предложенным по умолчанию именем `Subject1`, которое изменим на `Банкомат` (рис. 14.8).

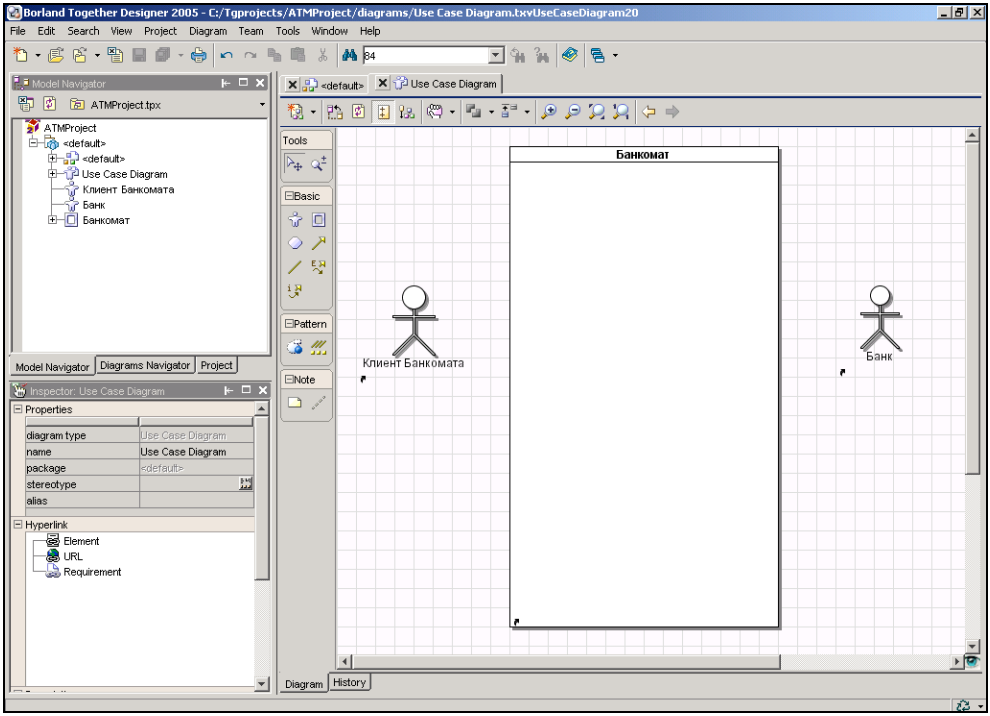


Рис. 14.8. Диаграмма вариантов использования после добавления на нее границы системы

14.1.4. Добавление вариантов использования

Для добавления варианта использования на диаграмму нужно нажать на специальной панели инструментов кнопку с изображением варианта использования и щелкнуть левой кнопкой мыши внутри границы системы. На диаграмме появится изображение варианта использования с маркерами изменения его геометрических размеров и предложенным по умолчанию именем, которое следует изменить (рис. 14.9).

Редактирование свойств варианта использования можно выполнять в окне инспектора. Аналогичным образом на данную диаграмму следует добавить следующие варианты использования: `Получение справки о состоянии счета`, `Проверка ПИН-кода` и `Блокирование кредитной карточки`.

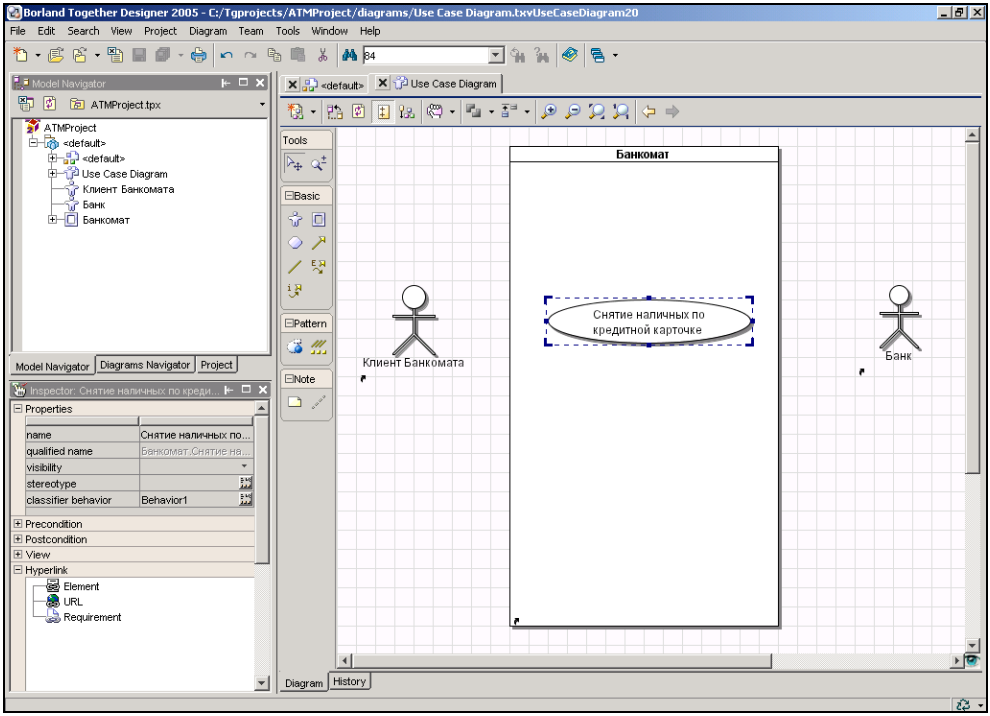


Рис. 14.9. Диаграмма вариантов использования после добавления на нее варианта использования

14.1.5. Добавление ассоциаций

Для добавления ассоциации между актером и вариантом использования на диаграмму нужно нажать на специальной панели инструментов кнопку с изображением пиктограммы ассоциации и отпустить левую кнопку мыши. После этого щелкнуть левой кнопкой мыши на изображении актера на диаграмме и отпустить ее на изображении варианта использования. В результате этих действий на диаграмме появится изображение ассоциации, которая соединяет актера с вариантом использования (рис. 14.10).

Дополнительно для данной ассоциации можно задать текстовый стереотип `<<communicate>>`, который уточняет характер этой ассоциации на диаграмме вариантов использования. С этой целью в окне инспектора свойств ассоциации следует щелкнуть на кнопке рядом со свойством **stereotype** (стереотип) и в открывшемся диалоговом окне щелкнуть на кнопке **Add** (Добавить). В появившемся поле ввода ввести текст: `communicate` (рис. 14.11) и нажать кнопку **OK**.

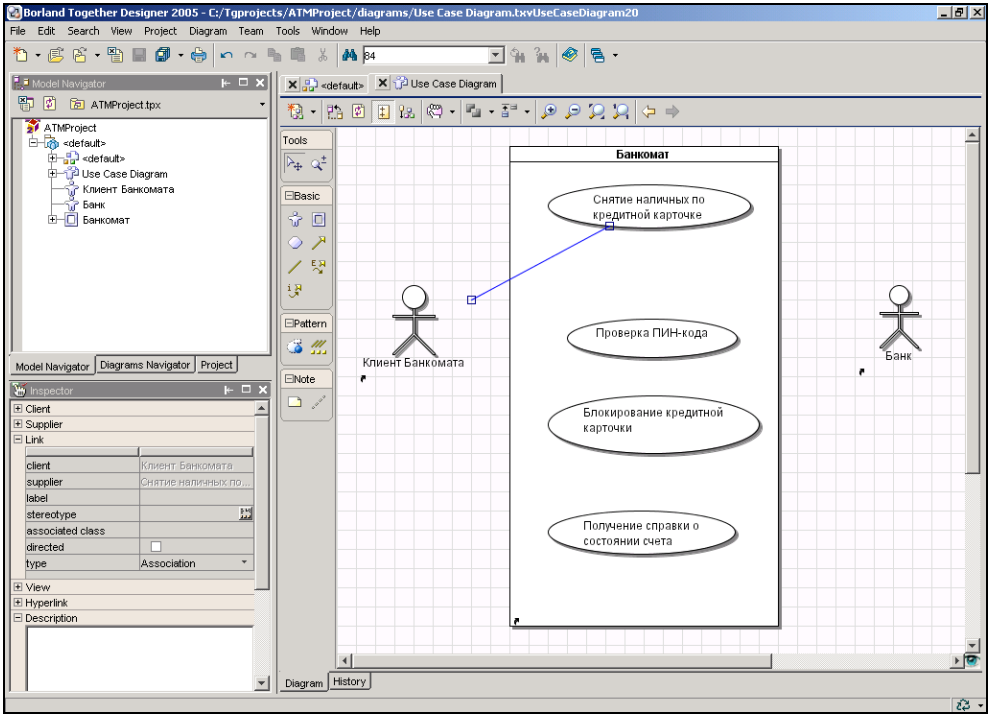


Рис. 14.10. Диаграмма вариантов использования после добавления на нее ассоциации

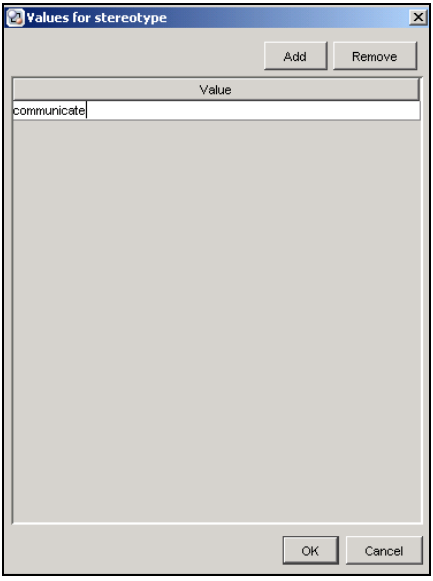


Рис. 14.11. Диалоговое окно задания текстового стереотипа для ассоциации

В результате выполненных действий на диаграмме рядом с линией ассоциации появится текстовый стереотип. Аналогичным образом следует добавить на диаграмму ассоциации между актером `Клиент Банкомата` и вариантом использования `Получение справки о состоянии счета`, а также между актером `Банк` и вариантами использования `Снятие наличных по кредитной карточке` и `Получение справки о состоянии счета`.

14.1.6. Добавление зависимостей

Для добавления зависимости со стереотипом `<<include>>` между двумя вариантами использования нужно нажать на специальной панели инструментов кнопку с изображением пиктограммы зависимости **Include** и отпустить левую кнопку мыши. После этого щелкнуть левой кнопкой мыши на изображении варианта использования `Снятие наличных по кредитной карточке` и отпустить ее на изображении варианта использования `Проверка ПИН-кода`. В результате этих действий на диаграмме появится изображение отношения зависимости со стереотипом `<<include>>`, которое соединяет два выбранных варианта использования (рис. 14.12).

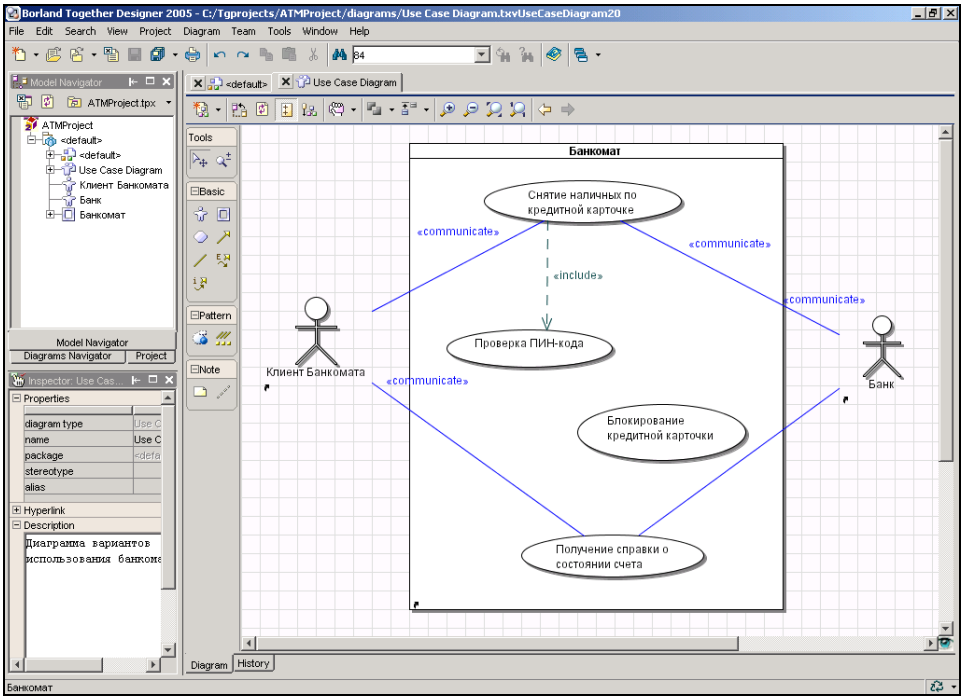


Рис. 14.12. Диаграмма вариантов использования после добавления на нее отношения зависимости со стереотипом `<<include>>`

Аналогичным образом следует добавить на диаграмму зависимость со стереотипом <<include>> между вариантом использования Получение справки о состоянии счета и вариантом использования Проверка ПИН-кода.

Для добавления зависимости со стереотипом <<extend>> между двумя вариантами использования нужно нажать на специальной панели инструментов кнопку с изображением пиктограммы зависимости **Extend** и отпустить левую кнопку мыши. После этого щелкнуть левой кнопкой мыши на изображении варианта использования Блокирование кредитной карточки и отпустить ее на изображении варианта использования Проверка ПИН-кода. В результате этих действий на диаграмме появится изображение отношения зависимости со стереотипом <<extend>>, которое соединяет два выбранных варианта использования (рис. 14.13).

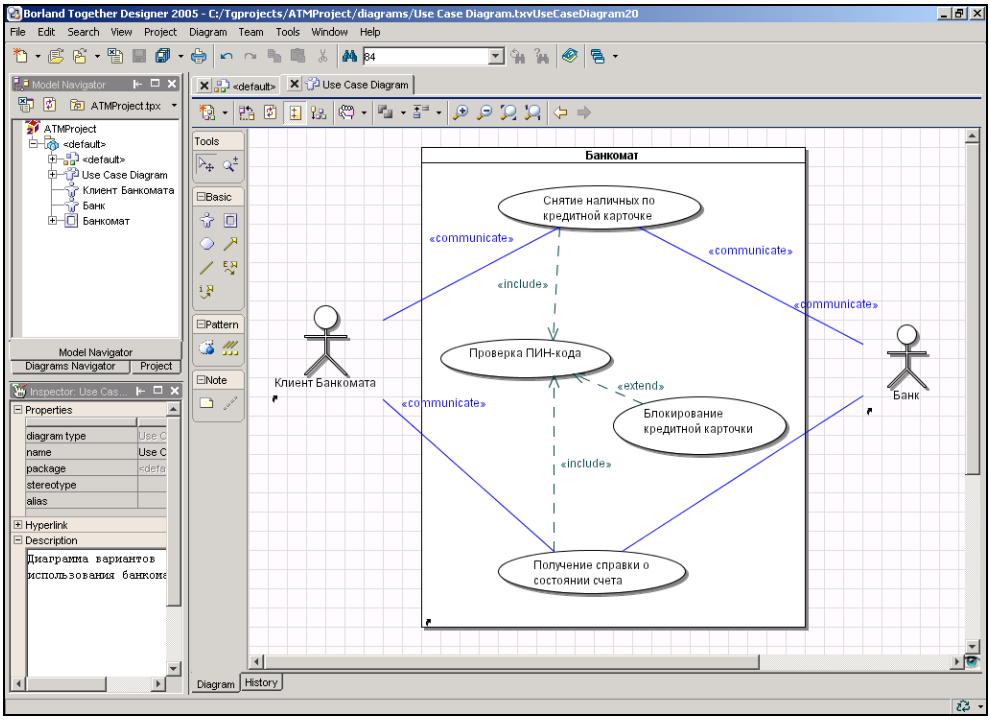


Рис. 14.13. Диаграмма вариантов использования после добавления на нее отношения зависимости со стереотипом <<extend>>

14.1.7. Добавление текстового файла со сценарием варианта использования

Для отдельных вариантов использования можно добавить ссылку на файл с текстом сценария, который описывает типичный ход событий выполнения соответствующего варианта использования. В качестве примера добавим к варианту использования `Снятие наличных по кредитной карточке` текстовый файл формата MS Word с описанием сценария его выполнения (см. табл. 3.1—3.3). Для этого необходимо выделить этот вариант использования в окне диаграммы и в окне инспектора раскрыть список свойств **Hiperlink** (Гиперссылка). Далее нужно выделить строку **URL** и выполнить операцию контекстного меню **New** (Новый). В появившемся окне выбрать нужный файл и щелкнуть на кнопке **OK**.

Аналогичные действия могут быть выполнены в контекстном меню выбранного варианта использования.

В результате этих действий в окне инспектора появится ниже строки **URL** изображение пиктограммы текстового файла (рис. 14.14). Двойной щелчок на этой пиктограмме позволяет открыть этот файл в соответствующем приложении (в рассматриваемом случае — в текстовом процессоре MS Word).

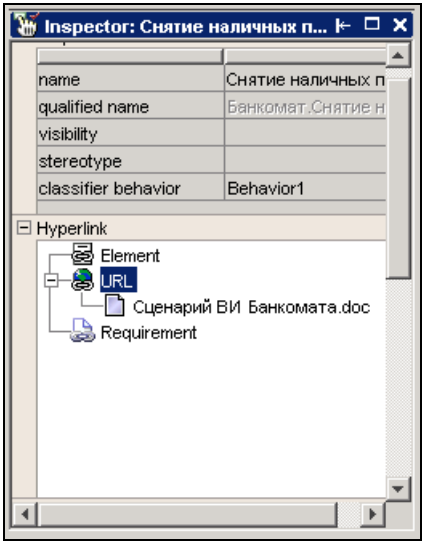


Рис. 14.14. Окно инспектора со ссылкой на файл с текстом сценария варианта использования

Напомним, что диаграмма вариантов использования является высокоуровневым представлением модели, поэтому она не должна содержать слишком много мелких вариантов использования или актеров. В ходе последующей работы над проектом построенная диаграмма может быть модифицирована посредством добавления новых элементов, таких как варианты использования и актеры, или удалением некоторых из них.

Для удаления графического элемента с диаграммы и из модели его следует выделить на диаграмме или в окне навигатора модели и нажать клавишу <Delete>. Для отмены удаления можно воспользоваться комбинацией клавиш <Ctrl>+<Z>.

При работе со связями на диаграмме вариантов использования следует помнить о назначении соответствующих отношений в нотации языка UML 2.0. Речь идет о том, что если для двух элементов выбранный вид связи не является допустимым, то в большинстве случаев программа Together Designer сообщит об этом разработчику, и такая связь не будет добавлена на диаграмму.

Разработку модели системы управления банкоматом продолжим построением следующего типа канонических диаграмм — диаграммы классов.

14.2. Разработка диаграммы классов в среде Together Designer

Диаграмма классов является основным логическим представлением модели и содержит детальную информацию о внутреннем устройстве объектно-ориентированной программной системы. Нотация элементов диаграммы классов и общие рекомендации по ее построению были рассмотрены в *главе 4*. Хотя диаграмма классов присутствует в проекте по умолчанию, для логического представления модели следует создать новую диаграмму этого типа.

Для разработки диаграммы классов рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены для диаграммы вариантов использования в *разд. 14.1*. В результате появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Class**. При этом в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы классов. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы классов (табл. 14.2). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 14.2. *Назначение кнопок специальной панели инструментов диаграммы классов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Package	Добавляет на диаграмму пакет
	Class	Добавляет на диаграмму класс
	Interface	Добавляет на диаграмму интерфейс
	Association Class	Добавляет на диаграмму ассоциацию-класс
	Port	Добавляет на диаграмму порт
	Instance Specification	Добавляет на диаграмму спецификацию экземпляра или объект
	Generalization/ Implementation	Добавляет на диаграмму отношение обобщения
	Provided Interface	Добавляет на диаграмму предоставляемый интерфейс
	Required Interface	Добавляет на диаграмму требуемый интерфейс
	Association	Добавляет на диаграмму ассоциацию
	Association End	Добавляет на диаграмму конец ассоциации
	Dependency	Добавляет на диаграмму отношение зависимости

Кроме этой панели инструментов для диаграммы классов могут быть добавлены еще две панели с дополнительными графическими стереотипами классов: **Business Process 2.0** и **Software Development Process**. Эти панели можно отобразить с помощью операции главного меню **Diagram | Properties** (Диаграмма | Свойства). Именно на этих панелях имеются кнопки для создания классов с известными стереотипами: класс-сущность, граничный класс и класс-контроллер. При необходимости читатели могут воспользоваться этой возможностью самостоятельно.

14.2.1. Добавление классов

Для добавления класса на диаграмму классов нужно нажать на специальной панели инструментов кнопку с изображением пиктограммы класса, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение класса с маркерами изменения его геометрических размеров и предложенным по умолчанию именем, которое следует изменить на `Устройство Чтения Карточки` (рис. 14.15).

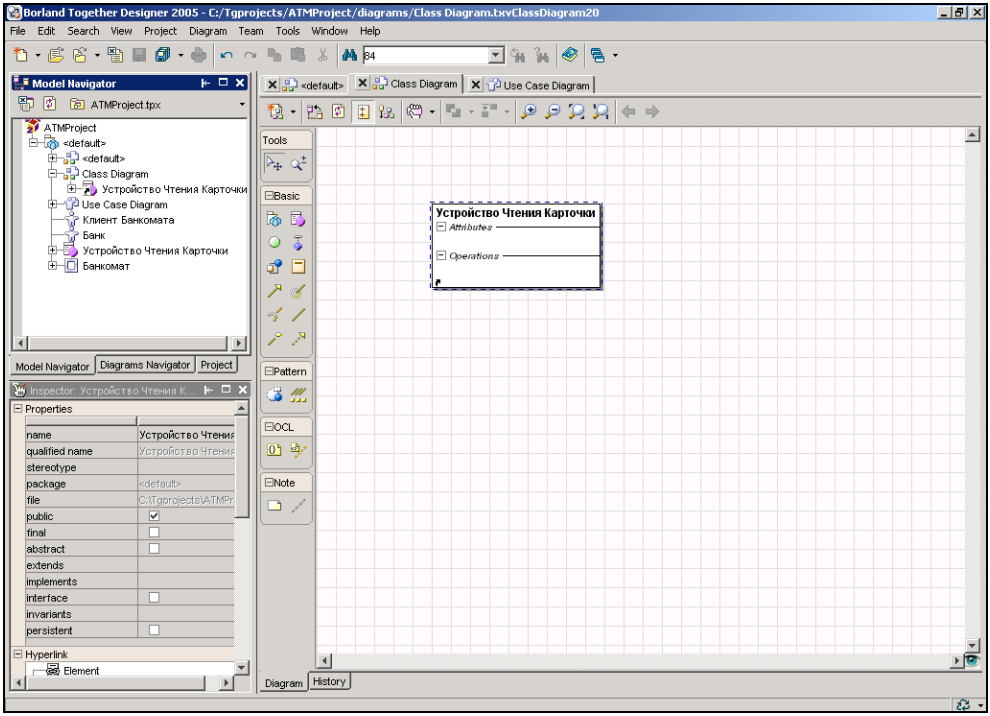


Рис. 14.15. Диаграмма классов после добавления на нее класса `Устройство Чтения Карточки`

Для рассматриваемой модели системы управления банкоматом на диаграмму классов следует добавить аналогичным образом следующие классы: `Контроллер Банкомата`, `Транзакция Банкомата`, `Контроллер Банка`, `Принтер`, `Устройство Выдачи Наличных`, `Клавиатура`, `Экран`, а также интерфейс `IКонтроллер Банка`.

14.2.2. Добавление атрибутов классов

Из всех графических элементов класс обладает максимальным набором свойств, главными из которых являются его атрибуты и операции. Добавить атрибут к созданному ранее классу можно с помощью операции контекстного меню **New | Attribute** (Новый | Атрибут). Эта операция доступна для класса, выделенного на диаграмме классов или в навигаторе модели. В этом случае активизируется строка ввода в области атрибутов соответствующего класса на диаграмме классов. Добавим рассмотренным способом атрибут с именем `идентификатор карточки` к классу `Транзакция Банкомата` (рис. 14.16).

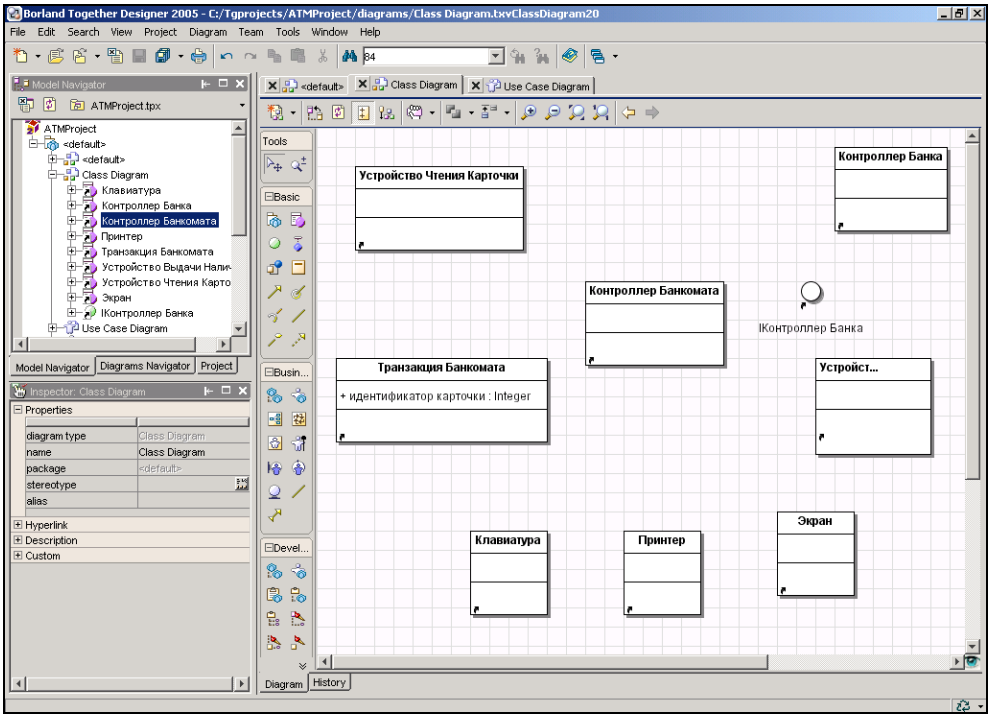


Рис. 14.16. Диаграмма классов после добавления на нее атрибута для класса `Транзакция Банкомата`

В качестве типа значений атрибута оставим предложенный по умолчанию тип `Integer`, а его видимость изменим на общедоступную (`public`). Редактирование свойств атрибута выполняется в окне инспектора, при этом каждый атрибут имеет собственный набор свойств (рис. 14.17).

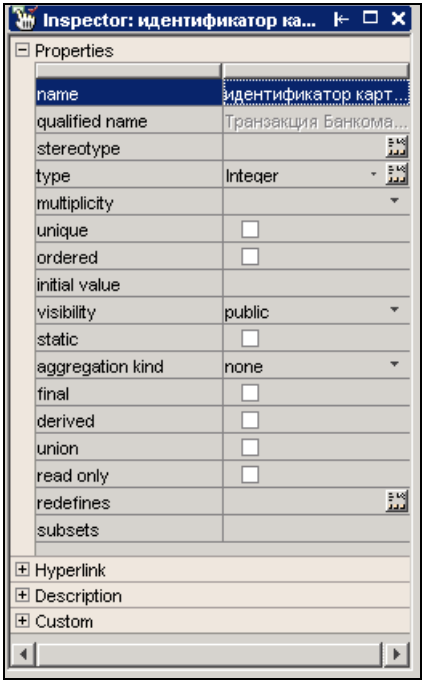


Рис. 14.17. Окно инспектора свойств атрибута

Для атрибутов классов в окне инспектора можно задавать также все остальные элементы описания, которые были рассмотрены в *главе 4* при описании общего формата записи атрибутов. Для рассматриваемой модели на диаграмму классов следует добавить аналогичным образом для класса `Транзакция Банкомата` следующие 3 атрибута: `значение ПИН-кода карточки`, `введенный ПИН-код` и `введенная сумма наличных`. У остальных классов модели атрибуты будут отсутствовать.

14.2.3. Добавление операций классов

Добавить операцию к созданному ранее классу можно с помощью контекстного меню **New | Operation** (Новый | Операция). Это действие доступно для класса, выделенного на диаграмме классов или в навигаторе модели. В этом случае активизируется строка ввода в области операций соответствующего класса на диаграмме классов. Добавим рассмотренным способом операцию с именем: `прочитать идентификатор карточки` к классу `Устройство Чтения Карточки` (рис. 14.18).

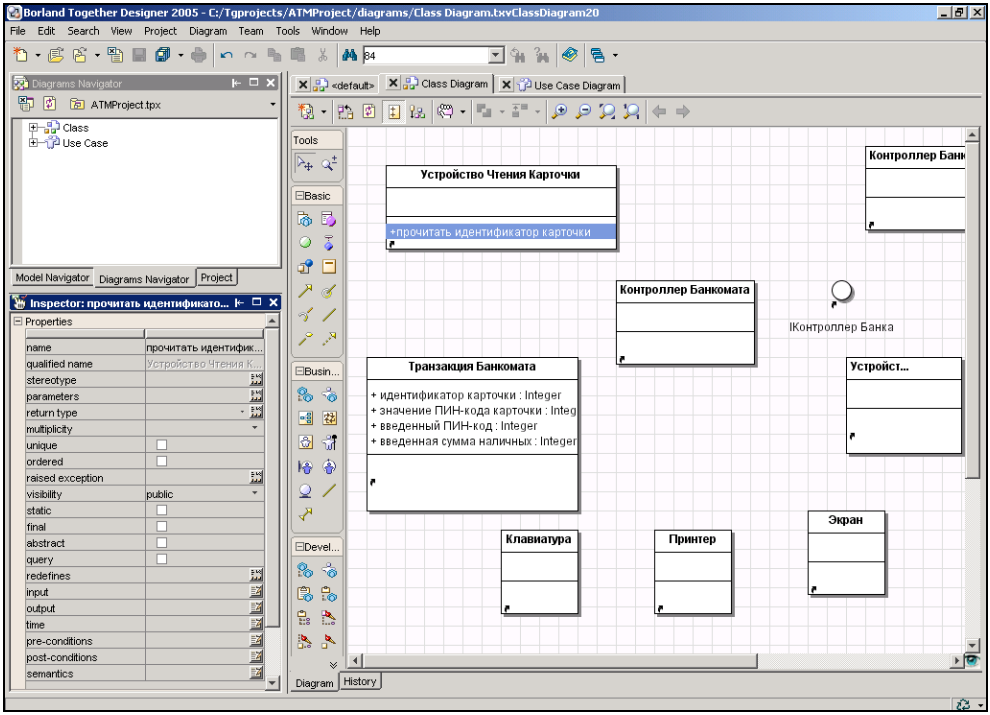


Рис. 14.18. Диаграмма классов после добавления на нее операции для класса `Устройство Чтения Карточки`

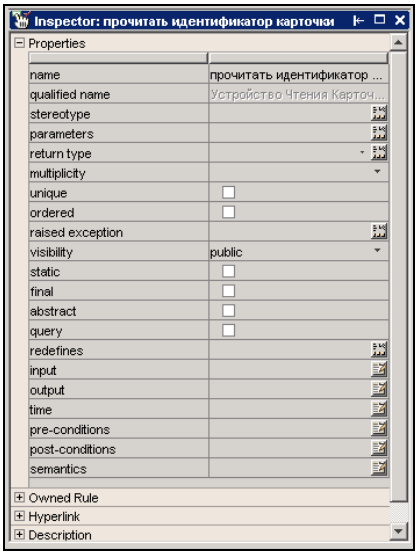


Рис. 14.19. Окно инспектора свойств операции

Напомним, что к имени операции справа добавляются круглые скобки, которые служат визуальным отличием операций от атрибутов. Значение видимости операции оставим предложенное по умолчанию — общедоступная (public). Редактирование свойств операции выполняется в окне инспектора, при этом каждая операция имеет собственный набор свойств (рис. 14.19).

Для операций классов в окне инспектора можно задавать также все остальные элементы описания, которые были рассмотрены в *главе 4* при описании общего формата записи операций. Для окончательного построения диаграммы классов рассматриваемой модели банкомата следует описанным выше способом добавить операции ко всем классам, а также специфицировать отдельные свойства этих операций.

С этой целью следует выполнить следующие действия:

1. Для класса Устройство Чтения Карточки добавить операцию прочитать ПИН-код() с видимостью public. В качестве типа возвращаемого результата для этой операции следует выбрать тип Integer (Целочисленный), а в секцию **Description** (Описание) данной операции ввести поясняющий текст: "Вызывается после того, как кредитная карточка вставлена в Устройство чтения карточки".
2. Для класса Устройство Чтения Карточки добавить операцию вернуть кредитную карточку() с видимостью public. В секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Вызывается после завершения транзакции".
3. Для класса Устройство Чтения Карточки добавить операцию блокировать кредитную карточку() с видимостью public. В секцию **Description** (Описание) данной операции ввести поясняющий текст: "Вызывается после того, как трижды введен неправильный ПИН-код или установлен факт утраты кредитной карточки владельцем".
4. Для класса Транзакция Банкомата добавить операцию создать новую транзакцию() с видимостью public. В секцию **Description** (Описание) данной операции ввести поясняющий текст: "Вызывается после того, как клиент вставил кредитную карточку в устройство чтения карточки".
5. Для класса Транзакция Банкомата добавить операцию проверить правильность ПИН-кода() с видимостью public. В качестве типа возвращаемого результата для этой операции следует выбрать тип Boolean (Логический), а в секцию **Description** (Описание) данной операции ввести поясняющий текст: "Вызывается после того, как клиент ввел значение ПИН-кода с клавиатуры банкомата".

6. Для класса `Транзакция Банкомата` добавить операцию `завершить транзакцию()` с видимостью `public`. В секцию **Description** (Описание) данной операции ввести поясняющий текст: "Вызывается после завершения всех действий банкомата по обслуживанию клиента".
7. Для класса `Экран` добавить операцию `показать меню опций()` с видимостью `public`.
8. Для класса `Экран` добавить операцию `показать меню снятия суммы()` с видимостью `public`.
9. Для класса `Клавиатура` добавить операцию `ввести ПИН-код()` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Integer`, а в секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Вызывается после окончания ввода значения ПИН-кода с клавиатуры".
10. Для класса `Клавиатура` добавить операцию `ввести тип транзакции()` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Boolean` (Логический), а в секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Возвращает значение Истина, если клиент выбирает снятие наличных, и значение Ложь, если клиент выбирает получение справки о состоянии счета".
11. Для класса `Клавиатура` добавить операцию `ввести сумму снятия наличных()` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Integer`, а в секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Вызывается после окончания ввода значения снимаемой суммы с клавиатуры".
12. Для класса `Устройство Выдачи Наличных` добавить операцию `выдать наличные()` с видимостью `public`. В секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Вызывается после того, как выполнено снятие запрошенной клиентом суммы со счета".
13. Для класса `Принтер` добавить операцию `распечатать чек()` с видимостью `public`. В секцию **Description** (Описание) данной операции следует ввести поясняющий текст: "Вызывается по дополнительному запросу клиента".
14. Для интерфейса `IИнтерфейс Банка` добавить операцию `проверить идентификатор карточки(идентификатор карточки: Integer)` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Boolean` (Логический), а в качестве ее параметра задать `идентификатор карточки`. Для задания параметра необходимо перейти в окно инспектора и выполнить щелчок на кнопке в строке **parameters**. В появившемся окне добавить новый параметр к операции и ввести его имя.

15. Для интерфейса `IИнтерфейс Банка` добавить операцию `открыть счет клиента(идентификатор карточки: Integer)` с видимостью `public`. В качестве ее параметра этой операции следует задать `идентификатор карточки`.

16. Для интерфейса `IИнтерфейс Банка` добавить операцию `проверить баланс клиента (идентификатор карточки: Integer, введенная сумма наличных: Integer)` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Boolean` (Логический). В качестве первого целочисленного аргумента этой операции следует задать `идентификатор карточки`, а в качестве второго аргумента — `введенная сумма наличных`.

17. Для интерфейса `IИнтерфейс Банка` добавить операцию `уменьшить счет клиента(идентификатор карточки: Integer, введенная сумма наличных: Integer)` с видимостью `public`. В качестве типа возвращаемого результата для этой операции следует выбрать тип `Boolean` (Логический). В качестве первого целочисленного аргумента этой операции следует задать `идентификатор карточки`, а в качестве второго аргумента — `введенная сумма наличных`.

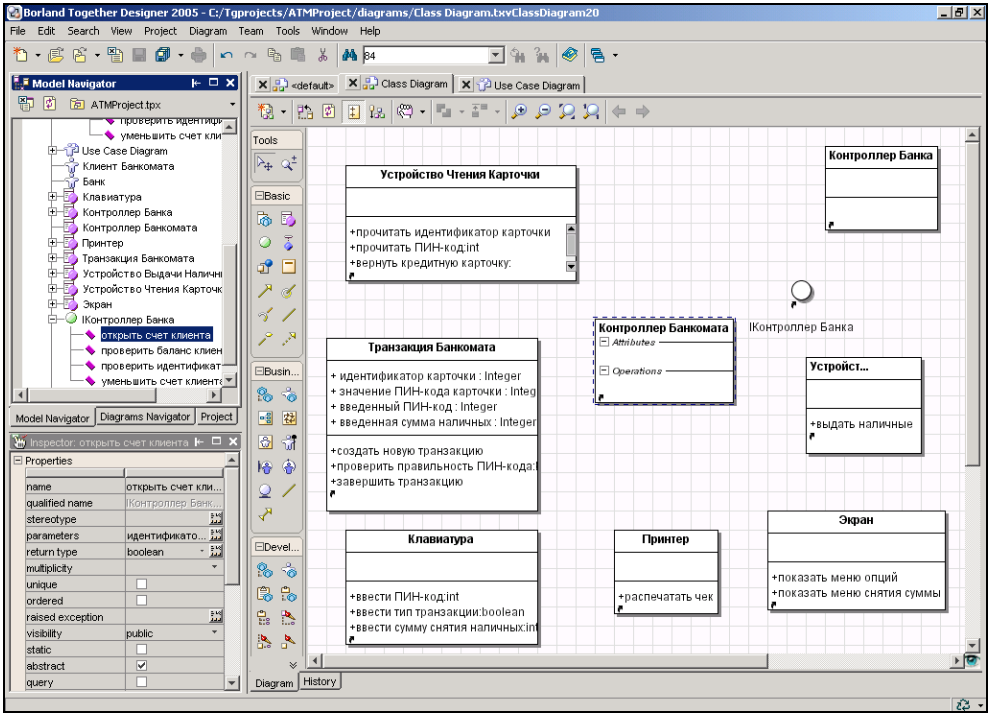


Рис. 14.20. Диаграмма классов после добавления на нее атрибутов и операций классов

Следует заметить, что поскольку интерфейс `IИнтерфейс Банка` изображен в форме круга, его операции не отображаются в этом символе. Поэтому для их добавления и последующего редактирования следует воспользоваться контекстным меню окна навигатора модели. Все операции этого интерфейса следует также определить и для класса `Контроллер Банка` с целью удобства последующей работы с диаграммами взаимодействия.

Выполнить эти действия предлагается читателям самостоятельно. Построенная в результате указанных действий диаграмма классов будет иметь вид, представленный на рис. 14.20.

14.2.4. Добавление отношений на диаграмму классов

Добавление на диаграмму классов отношений между классами типа ассоциации, зависимости, агрегации, композиции, реализации и обобщения выполняется следующим образом. На специальной панели инструментов выбирается требуемый тип отношения щелчком по кнопке с соответствующим изображением. Если отношение направленное, то на диаграмме классов надо выделить первый элемент отношения (источник или клиент, от которого исходит стрелка) и, не отпуская нажатую левую кнопку мыши, переместить ее указатель ко второму элементу отношения (приемник или цель, к которой направлена стрелка). После перемещения ко второму элементу кнопку мыши следует отпустить, в результате чего на диаграмму классов будет добавлено новое отношение.

На специальной панели инструментов диаграммы классов отсутствует кнопка с пиктограммой агрегации. В этом случае необходимо предварительно добавить на диаграмму ассоциацию и в инспекторе свойств выбрать тип **Agregation** или **Composition** из вложенного списка в строке **type** (тип). Если отношение направленное, то в инспекторе свойств следует выставить отметку у свойства **derected** (направленное). Для отношений можно определить кратность каждого из концов, задать имя и стереотип, использовать ограничения и роли `Клиента` и `Цели`, а также некоторые другие свойства.

Для окончательного построения диаграммы классов рассматриваемого примера следует описанным выше способом добавить следующие ассоциации:

1. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `Устройство чтения карточки`. В качестве кратности концов этой ассоциации установить значение 1.
2. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `Принтер`. В качестве кратности концов этой ассоциации установить значение 1.

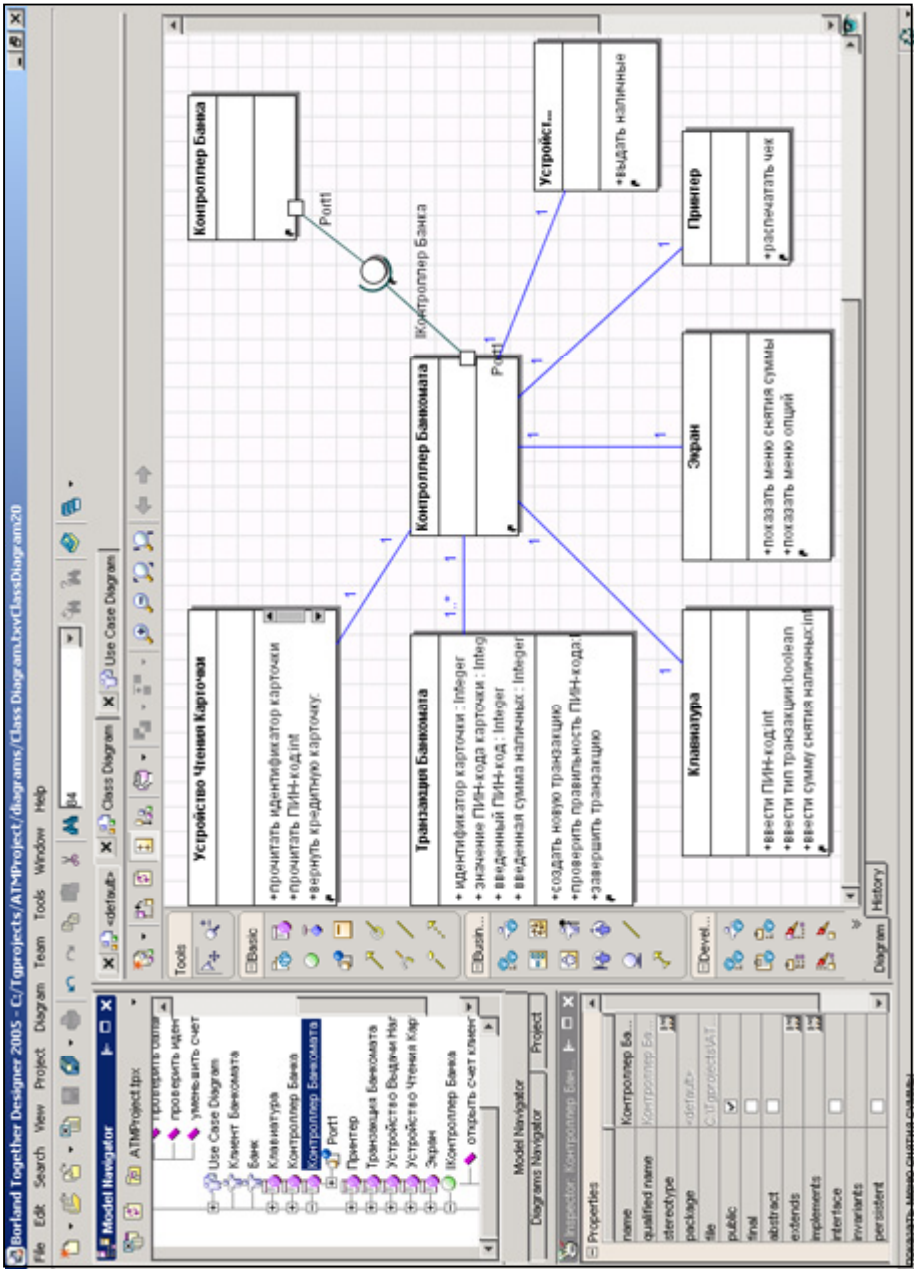


Рис. 14.21. Окончательный вид диаграммы классов разрабатываемой модели управления банкоматом

3. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `Клавиатура`. В качестве кратности концов этой ассоциации установить значение 1.
4. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `Устройство выдачи наличных`. В качестве кратности концов этой ассоциации установить значение 1.
5. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `Экран`. В качестве кратности концов этой ассоциации установить значение 1.
6. Добавить ассоциацию от класса `Контроллер Банкомата` к классу `IКонтроллер Банка`. В качестве кратности конца этой ассоциации для первого класса установить значение `0..n`, а кратности конца ассоциации для второго класса установить значение 1.
7. Добавить связь **Required Interface** (Требуемый интерфейс) от класса `Контроллер Банкомата` к интерфейсу `IКонтроллер Банка`. Поскольку коммуникация между банкоматом и банком осуществляется по специальному протоколу, необходимо предварительно добавить порт для класса `Контроллер Банкомата`, а соответствующую связь провести от этого порта к интерфейсу.
8. Добавить связь **Provided Interface** (Предоставляемый интерфейс) от класса `Банк` к интерфейсу `IКонтроллер Банка`. С этой целью необходимо предварительно добавить порт для класса `Банк`, а соответствующую связь провести от этого порта к интерфейсу.

Построенная таким образом диаграмма классов будет иметь вид, представленный на рис. 14.21.

На построенной диаграмме классов приведен вариант изображения предоставляемого и требуемого интерфейсов в форме нотации "шара и шарнира".

14.3. Разработка диаграммы композитной структуры в среде Together Designer

Нотация элементов диаграммы композитной структуры и общие рекомендации по ее построению были рассмотрены в *главе 5*. Для разработки диаграммы композитной структуры рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены для диаграммы вариантов использования в *разд. 14.1*. В результате появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Composite Structure**.

В окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы композитной структуры. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы композитной структуры (табл. 14.3). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 14.3. *Назначение кнопок специальной панели инструментов диаграммы композитной структуры*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Class	Добавляет на диаграмму класс
	Interface	Добавляет на диаграмму интерфейс
	Collaboration	Добавляет на диаграмму кооперацию
	Collaboration Occurrence	Добавляет на диаграмму экземпляр кооперации
	Part	Добавляет на диаграмму часть
	Referenced Part	Добавляет на диаграмму ссылку на часть
	Port	Добавляет на диаграмму порт
	Provided Interface	Добавляет на диаграмму предоставляемый интерфейс
	Required Interface	Добавляет на диаграмму требуемый интерфейс
	Connector	Добавляет на диаграмму соединитель
	Collaboration Role	Добавляет на диаграмму роль кооперации
	Role Binding	Добавляет на диаграмму связывание роли

14.3.1. Добавление классов и частей

Для рассматриваемой модели системы управления банкоматом на диаграмме композитной структуры можно представить внутреннюю структуру класса `Банкомат`, который состоит из отдельных частей или устройств, моделируемых уже созданными классами. При этом дополнительно можно указать порты для связи банкомата с другими классами.

На первом этапе добавим на диаграмму композитной структуры 2 класса с именами: `Банкомат` и `Банк`. Для этого их следует просто перетащить из навигатора модели на рабочий лист диаграммы. Затем внутрь символа класса `Банкомат` добавим аналогичным образом следующие его классы в форме частей: `Устройство Чтения Карточки`, `Устройство Выдачи Наличных`, `Принтер`, `Клавиатура`, `Экран`, перетащив их из навигатора модели. Перетащим также на диаграмму интерфейс `IКонтроллер Банка`. Построенная таким образом диаграмма композитной структуры будет иметь вид, представленный на рис. 14.22.

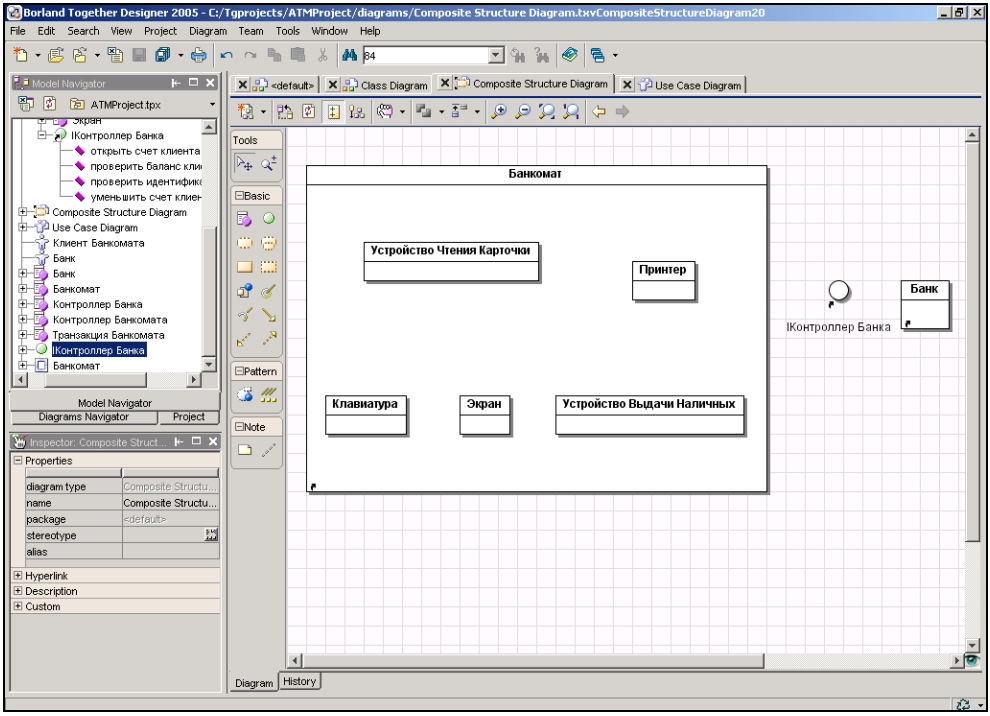


Рис. 14.22. Диаграмма композитной структуры после добавления классов и частей

14.3.2. Добавление портов и интерфейсов

Особенностью диаграммы композитной структуры является тот факт, что на ней можно показать все порты, необходимые для связи композитного класса с его окружением, а также внутри него с помощью соединителей. Для разрабатываемой модели на диаграмму композитной структуры можно добавить порты к классам `Банкомат` и `Банк`, а также ко всем частям класса `Банкомат`, понимая под этим наличие специальных особенностей для коммуникации этих элементов модели, как между собой, так и со средой. Дополнительно добавим 5 новых интерфейсов для каждой из частей с именами: `IКарточка`, `IКлавиатура`, `IПринтер`, `IНаличные` и `IЭкран`.

Для более удобного расположения портов их можно перемещать по границе символа класса, удерживая нажатой левую кнопку мыши на их изображении. Построенная таким образом диаграмма композитной структуры будет иметь вид, приведенный на рис. 14.23.

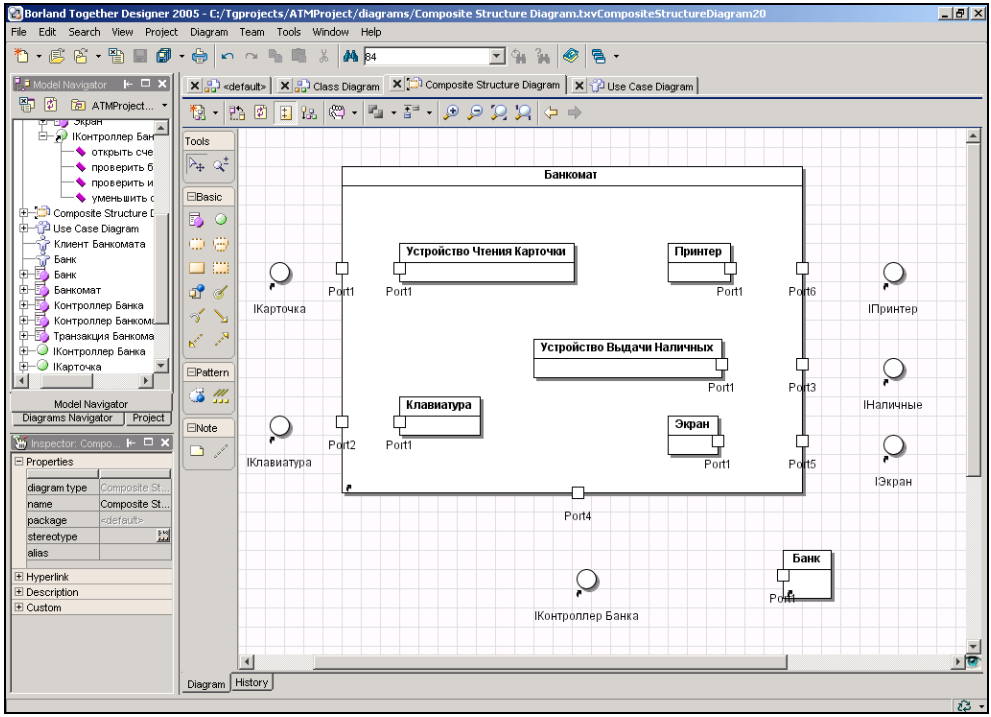


Рис. 14.23. Диаграмма композитной структуры после добавления новых интерфейсов и портов

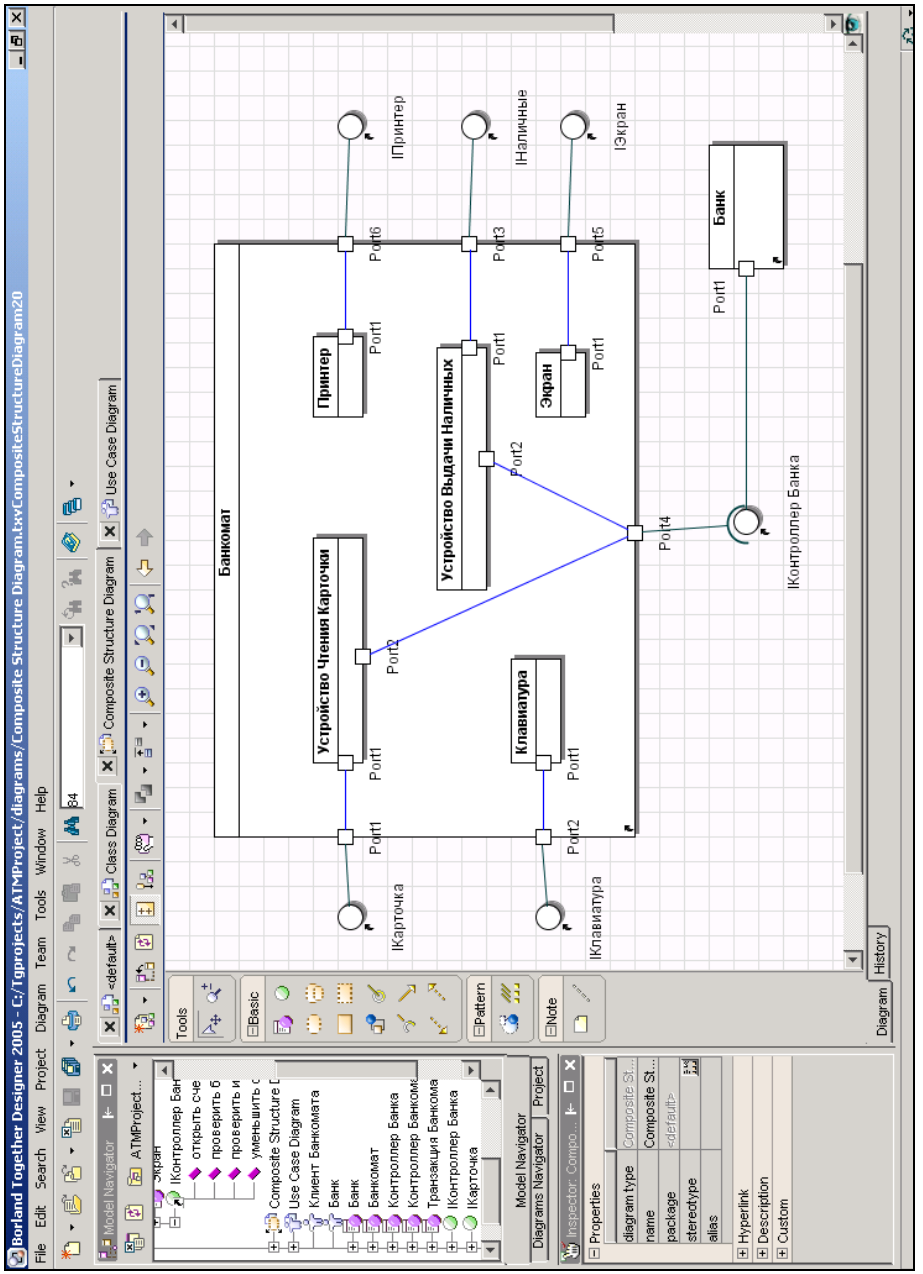


Рис. 14.24. Окончательный вид диаграммы композитной структуры разрабатываемой модели управления банкоматом

14.3.3. Добавление отношений на диаграмму композитной структуры

Добавление на диаграмму композитной структуры отношений между классами и частями выполняется аналогично добавлению отношений для других типов диаграмм. Для разрабатываемой модели на диаграмму композитной структуры следует добавить 5 соединителей (**Connector**) между портами класса `Банкомат` и портами его внутренних частей, а также 2 соединителя между портом `Port4` `Банкомата` и портами `Port2` `Устройства Чтения Карточки` и `Устройства Выдачи Наличных`. Следует добавить также 5 связей типа **Provided Interface** (Предоставляемый интерфейс) между портами `Банкомата` и соответствующими интерфейсами устройств, а также аналогичную связь от порта `Банка` к интерфейсу `IКонтроллер Банка`. Как и на диаграмме классов, интерфейс `IКонтроллер Банка` является требуемым для `Банкомата`, для чего необходимо провести на диаграмме соответствующую связь.

Построенный таким образом окончательный вариант диаграммы композитной структуры будет иметь вид, приведенный на рис. 14.24.

Вопрос о том, включать или не включать в композитную структуру `Банкомата` классы `Контроллер Банкомата` и `Транзакция Банкомата` предлагается решить читателям самостоятельно. В случае его положительного решения можно дополнить диаграмму указанными частями в качестве упражнения.

14.4. Разработка диаграммы последовательности в среде Together Designer

Диаграмма последовательности является формой визуализации взаимодействий в системе, на которой представляются линии жизни и сообщения между ними. Новыми элементами языка UML 2.0 являются фреймы взаимодействия и комбинированные фрагменты. Эти конструкции позволяют разработать одну диаграмму последовательности для реализации нескольких или даже всех вариантов использования модели, чего добиться в нотации языка UML 1.*x* было принципиально невозможно.

Нотация элементов диаграммы последовательности и общие рекомендации по ее построению были рассмотрены в *главе 7*. Для разработки диаграммы последовательности рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены для диаграммы вариантов использования в *разд. 14.1*. В результате отобразится окно с галереей

объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Sequence**. При этом в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы последовательности. Специальная панель инструментов будет содержать кнопки с изображениями пиктограмм элементов, необходимых для разработки диаграммы последовательности (табл. 14.4). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 14.4. *Назначение кнопок специальной панели инструментов диаграммы последовательности*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Lifeline	Добавляет на диаграмму линию жизни
	Execution Specification	Добавляет на диаграмму спецификацию выполнения
	Message	Добавляет на диаграмму сообщение
	State Invariant	Добавляет на диаграмму инвариант состояния
	Combined Fragment	Добавляет на диаграмму комбинированный фрагмент
	Interaction Use	Добавляет на диаграмму использование взаимодействия
	Tie Frame	Добавляет к линии жизни фрейм

14.4.1. Добавление линий жизни

Добавить линию жизни на диаграмму последовательности можно стандартным образом с помощью соответствующей кнопки на специальной панели инструментов. В результате выполнения действий по созданию линии жизни на диаграмме последовательности появится ее изображение с маркерами изменения геометрических размеров и вертикальной пунктирной линией, выходящей от символа этой линии жизни.

Имя типа добавленной линии жизни следует выбрать из списка существующих классов в дополнительном диалоговом окне, вызываемом при щелчке на кнопке в строке **type** (тип) в окне инспектора свойств. После выбора имени

типа `Контроллер Банкомата` оно будет указано после двоеточия. Собственное имя линии жизни для удобства визуализации модели можно удалить. Аналогично может быть добавлена на диаграмму вторая линия жизни с именем типа `Устройство Чтения Карточки`. Диаграмма последовательности после добавления второй линии жизни с открытым окном выбора типа будет иметь вид, представленный на рис. 14.25.

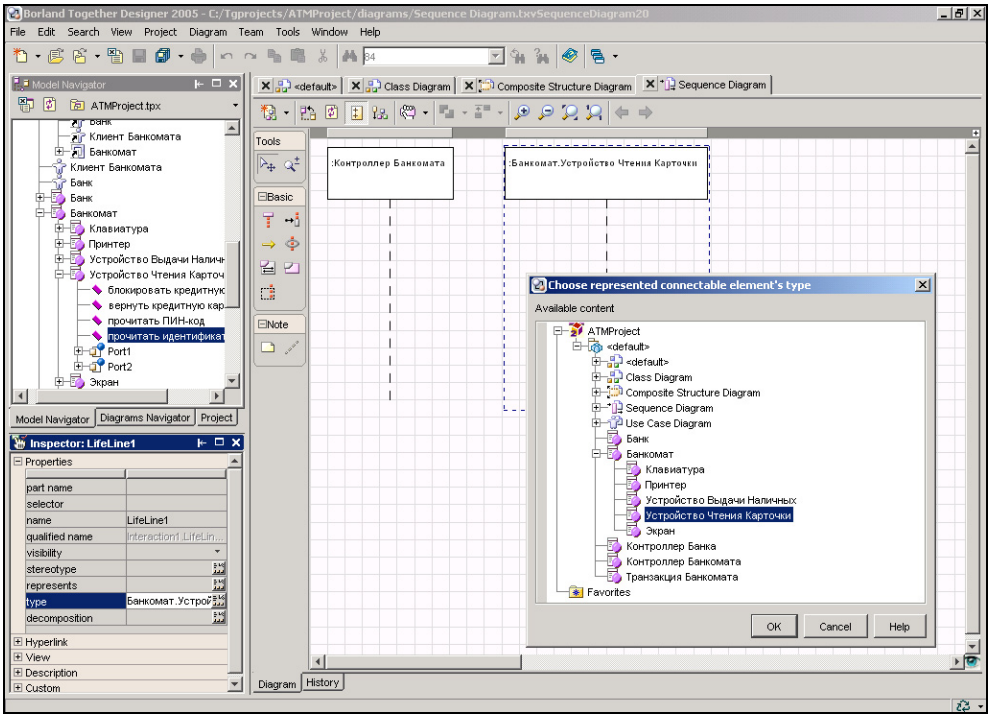


Рис. 14.25. Диаграмма последовательности после добавления двух линий жизни

Для рассматриваемой модели системы управления банкоматом на диаграмму последовательности следует добавить аналогичным образом линии жизни следующих типов: `Транзакция Банкомата`, `Контроллер Банка`, `Устройство Выдачи Наличных`, `Клавиатура`, `Принтер` и `Экран`.

14.4.2. Добавление сообщений

Для добавления сообщения между предварительно размещенными на диаграмме линиями жизни нужно нажать кнопку с изображением сообщения на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении одной линии жизни на диаграмме и

отпустить ее на изображении второй линии жизни. Имя сообщения выбирается из списка операций у класса, который является типом принимающей линии жизни. При необходимости можно задать новую операцию и добавить ее к описанию соответствующего класса.

В качестве первого сообщения добавим сообщение от линии жизни `Контроллер Банкомата` к линии жизни `Устройство Чтения Карточки`. Имя этого сообщения — "прочитать идентификатор карточки" — следует выбрать из списка операций класса `Устройство Чтения Карточки` в диалоговом окне, вызываемом при нажатии на кнопку в строке **signature** (сигнатура) в окне инспектора свойств этого сообщения. В результате выполнения этих действий на диаграмме последовательности появится изображение сообщения с порядковым номером, задаваемым по умолчанию (рис. 14.26). Для удобства визуализации и навигации по диаграмме можно воспользоваться специальным окном просмотра диаграммы (**Overview**), которое может быть открыто при щелчке на символе "глазок" в правом нижнем углу окна диаграммы. Перемещение прямоугольника со специальным цветовым фоном в окне просмотра позволяет быстро переместиться в любой регион диаграммы.

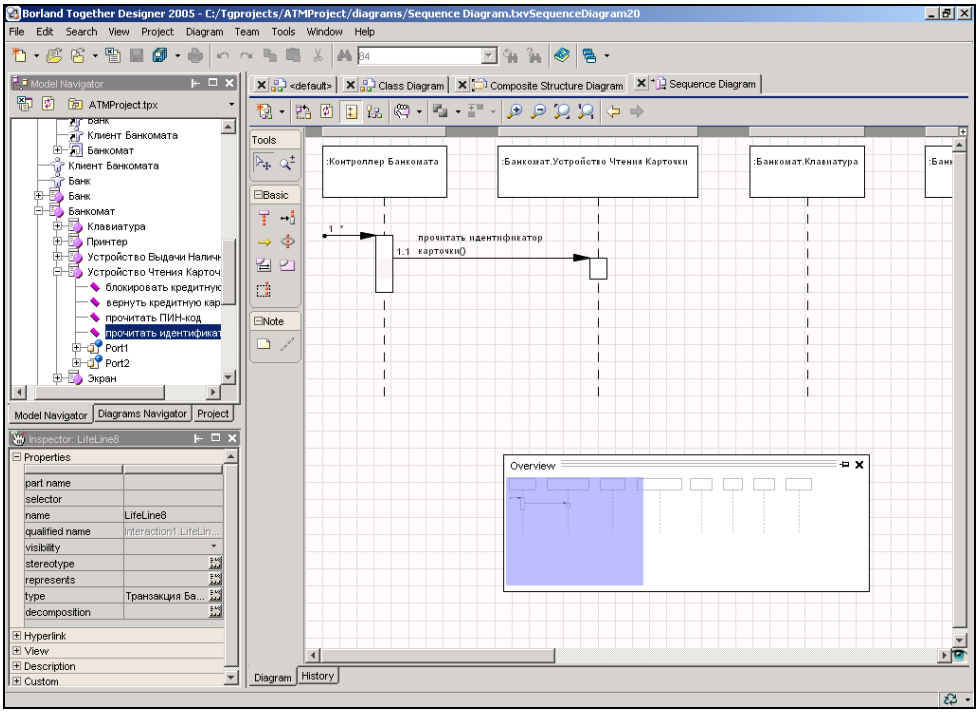


Рис. 14.26. Диаграмма последовательности после добавления сообщения с дополнительным окном обзора диаграммы

После этого следует добавить на диаграмму следующие сообщения:

- сообщение с именем "создать новую транзакцию", передаваемое от линии жизни с типом `Контроллер Банкомата` к линии жизни с типом `Транзакция Банкомата`;
- сообщение с именем "проверить идентификатор карточки", передаваемое от линии жизни с типом `Контроллер Банкомата` к линии жизни с типом `Контроллер Банка`.

Далее при отрицательном результате проверки идентификатора карточки типичный ход событий может прерваться. Это может быть представлено в форме специального комбинированного фрагмента.

14.4.3. Добавление комбинированных фрагментов

Для добавления комбинированного фрагмента следует воспользоваться специальной кнопкой на панели инструментов. После чего необходимо специфицировать операцию этого фрагмента и его операнды. Перечень всех операций комбинированных фрагментов в Together Designer, их теги и перевод представлены в табл. 14.5. Этот список совпадает с соответствующими операциями, определенными в спецификации языка UML 2.0.

Таблица 14.5. *Операторы взаимодействия на диаграмме последовательности*

Тег	Название	Перевод
`alt`	Alternatives	Альтернативы
`assert`	Assertion	Утверждение
`break`	Break	Завершение
`consider`	Consider	Рассмотрение
`critical`	Critical Region	Критический регион
`ignore`	Ignore	Игнорирование
`loop`	Loop	Цикл
`neg`	Negative	Отрицание
`opt`	Option	Необязательный
`par`	Parallel	Параллельный
`seq`	Weak Sequencing	Слабое следование
`strict`	Strict Sequencing	Строгое следование
	Custom	Произвольный (дополнительный в Together Designer)

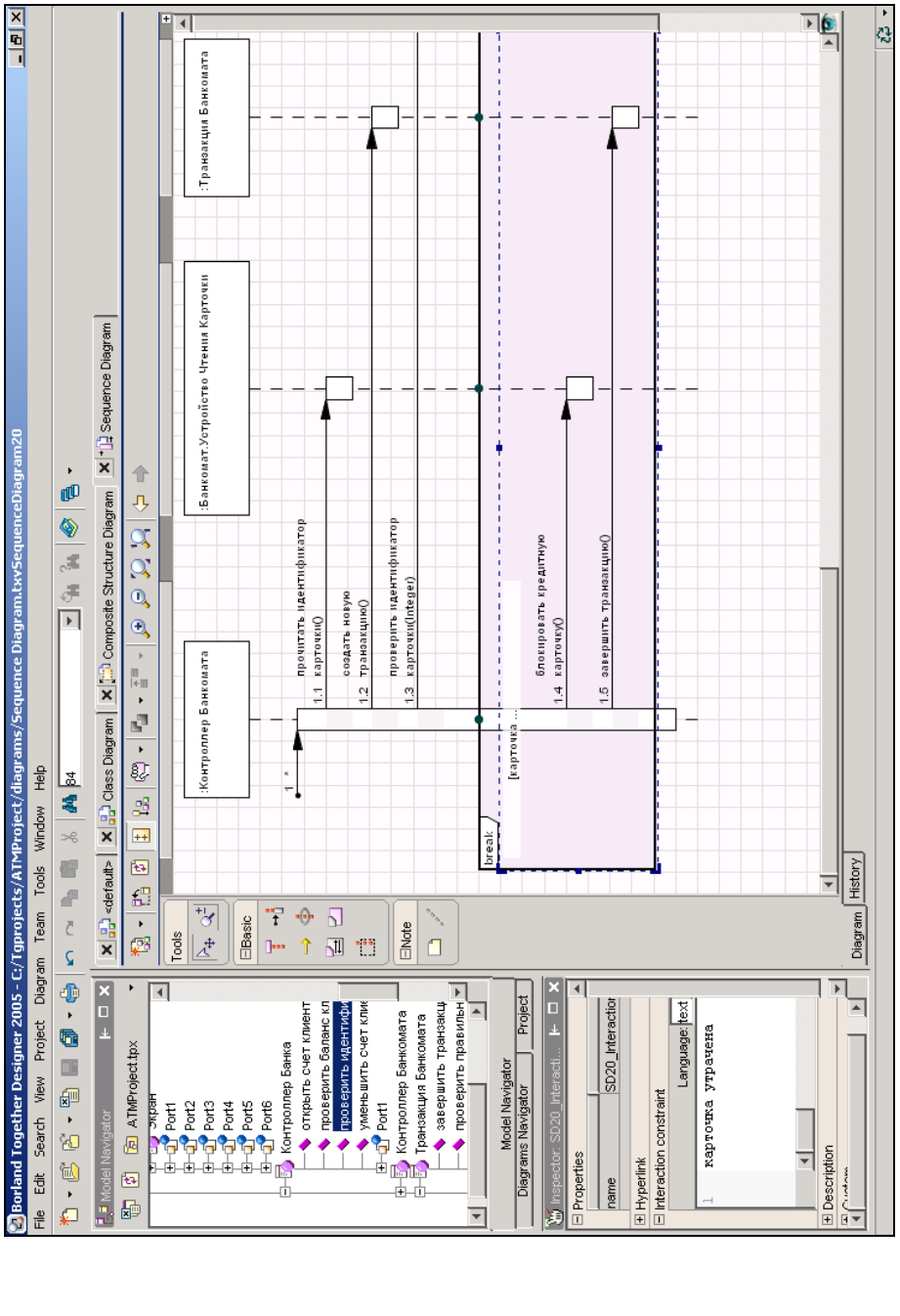


Рис. 14.27. Диаграмма последовательности после добавления комбинированного фрагмента `Завершение` с сообщениями

В разрабатываемую модель на данном этапе следует добавить комбинированный фрагмент Завершение. В этом комбинированном фрагменте выбрать тег операции Break (Завершение) и выделить в прямоугольнике фрейма ее операнд. Операнды отделяются друг от друга пунктирными линиями, но для данного фрагмента он единственный. Для операнда в окне инспектора необходимо задать ограничение взаимодействия, которое, по сути, представляет собой сторожевое условие на его входе.

Это условие вводится в поле **Interaction constraint** обычным текстом: карточка утрачена. Если оно выполняется, то по семантике фрагмента Завершение он будет выполняться. В противном случае он будет полностью проигнорирован, и управление перейдет к следующему сообщению, изображенному ниже этого фрагмента.

Далее в операнд этого фрагмента следует добавить два сообщения:

- сообщение с именем "блокировать кредитную карточку", передаваемое от линии жизни с типом Контроллер Банкомата к линии жизни с типом Устройство Чтения Карточки;
- сообщение с именем "завершить транзакцию", передаваемое от линии жизни с типом Контроллер Банкомата к линии жизни с типом Транзакция Банкомата.

В результате выполнения этих действий на диаграмме последовательности появится изображение комбинированного фрагмента Завершение с соответствующими сообщениями (рис. 14.27).

Ниже комбинированного фрагмента Завершение следует добавить комбинированный фрагмент Цикл со следующими сообщениями:

- сообщение с именем "прочитать ПИН-код", передаваемое от линии жизни с типом Контроллер Банкомата к линии жизни с типом Устройство Чтения Карточки;
- сообщение с именем "ввести ПИН-код", передаваемое от линии жизни с типом Контроллер Банкомата к линии жизни с типом Клавиатура;
- сообщение с именем "проверить правильность ПИН-кода", передаваемое от линии жизни с типом Контроллер Банкомата к линии жизни с типом Транзакция Банкомата.

В качестве аргумента цикла ввести (1,3), а в качестве ограничения взаимодействия — логическое выражение: result=false. Интерпретация этого фрагмента рассматривается в *главе 7*, поэтому здесь не приводится. В результате выполнения этих действий на диаграмме последовательности появится изображение комбинированного фрагмента Цикл с соответствующими сообщениями (рис. 14.28). Здесь для удобства визуализации указаны имена линий жизни в верхней области окна диаграммы, для чего следует нажать на небольшой значок (+) в верхнем правом углу этого окна.

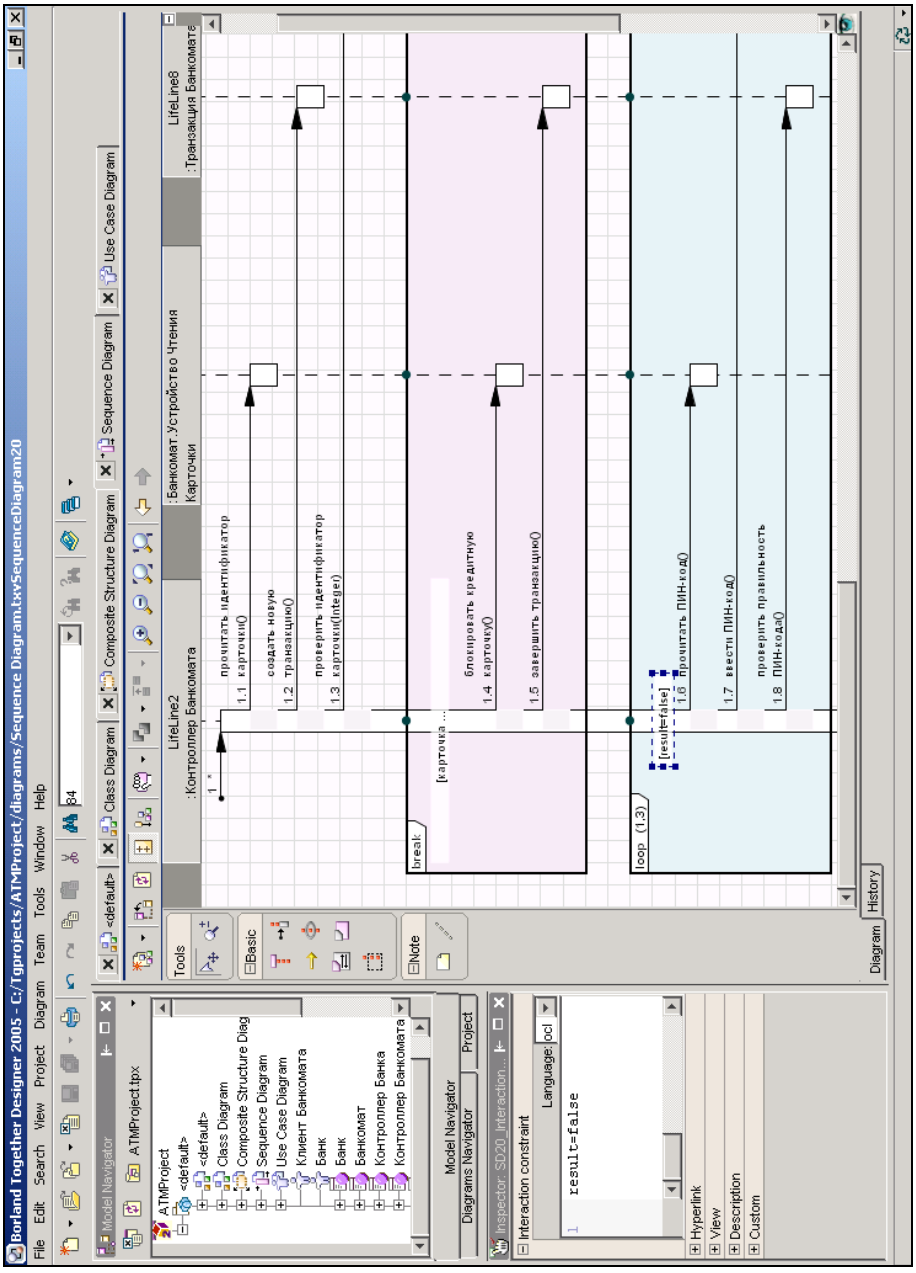


Рис. 14.28. Диаграмма последовательности после добавления комбинированного фрагмента `Цикл` с сообщениями

Следуя принятому сценарию работы банкомата, после трехкратно неверно введенного ПИН-кода нужно блокировать кредитную карточку. Это можно представить в модели с помощью второго комбинированного фрагмента `Завершение`, который следует добавить ниже фрагмента `Цикл`. Сообщения второго фрагмента `Завершение` будут идентичны первому, а в качестве ограничения взаимодействия можно указать выражение OCL: `result=false`.

После этого могут следовать сообщение экрану с именем операции `показать меню опций` и сообщение клавиатуре с именем операции `ввести тип транзакции`.

Далее можно приступать к моделированию основной функциональности. Это удобно сделать с помощью комбинированного фрагмента `Альтернативы` с двумя операндами. Первый будет выполняться при выборе клиентом снятия наличных, а второй — при выборе получения справки о состоянии счета. В первый операнд придется поместить вложенный комбинированный фрагмент `Завершение`, который завершит транзакцию и вернет карточку клиенту в случае, если запрошенная сумма превысит остаток на его счете. Понадобится и еще один фрагмент `Необязательный`, для выполнения операции печати чека при снятии наличных.

Второй операнд будет выполняться при выборе клиентом получения справки о состоянии счета. Здесь никаких вложенных фрагментов не будет, поскольку все представляется относительно тривиальным.

Для завершения построения диаграммы последовательности рассматриваемого проекта следует добавить указанные фрагменты и сообщения. Выполнение этих действий по построению окончательного варианта диаграммы последовательности предлагается выполнить читателям самостоятельно в качестве упражнения. В этом случае читатели наверняка получат эстетическое удовлетворение от изящества моделирования взаимодействий с помощью фреймов, что было недоступно при использовании предыдущих версий языка UML 1.*x*.

14.5. Разработка диаграммы коммуникации в среде Together Designer

Диаграмма коммуникации является разновидностью диаграммы взаимодействия и в контексте языка UML описывает динамический аспект взаимодействия объектов при реализации отдельных вариантов использования. Нотация элементов диаграммы коммуникации и общие рекомендации по ее построению были рассмотрены в *главе 9*. Для разработки диаграммы коммуникации рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать одним из способов, кото-

рые были рассмотрены для диаграммы вариантов использования в *разд. 14.1*. В результате появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Communication**.

В окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы коммуникации. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы коммуникации (табл. 14.6). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 14.6. *Назначение кнопок специальной панели инструментов диаграммы коммуникации*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Lifeline	Добавляет на диаграмму линию жизни
	Message	Добавляет на диаграмму сообщение

В общем случае работа с диаграммой коммуникации состоит в добавлении или удалении линий жизни и сообщений, а также заданием и изменением их свойств. Наличие на диаграмме коммуникации только двух типов элементов еще раз подчеркивает ее вспомогательный характер, что и было отмечено в *главе 9*.

В этом контексте нельзя не отметить уникальную возможность, которой обладает программа IBM Rational Rose. Хотя в ней реализована поддержка предыдущих версий языка UML 1.*x*, но все изменения, вносимые в диаграмму, последовательности, автоматически вносятся в диаграмму кооперации, и наоборот. Для переключения между этими диаграммами взаимодействия нужно просто нажать "волшебную" клавишу <F5>. Поскольку в языке UML 2.0 графические возможности диаграммы последовательности намного превосходят возможности диаграммы коммуникации, подобное переключение между диаграммами стало потенциально невозможным. По этой причине ее нужно строить разработчику самостоятельно в качестве отдельного этапа работы над проектом.

В качестве примера будет построена диаграмма коммуникации для типичного хода событий при реализации варианта использования `Снятие наличных по кредитной карточке`. Напомним, что типичный ход событий не учитывает обмен сообщениями при наступлении исключений или прерываний в моделируемом поведении системы. Все подобные исключения могут быть представлены на отдельных диаграммах коммуникации.

14.5.1. Добавление линий жизни

Линии жизни на диаграмму коммуникации добавляются стандартным образом с помощью соответствующей кнопки на специальной панели инструментов. В результате выполнения действий по созданию линии жизни на диаграмме коммуникации появится ее изображение в форме прямоугольника с маркерами изменения геометрических размеров.

Имя типа добавленной линии жизни следует выбрать из списка существующих классов в дополнительном диалоговом окне, вызываемом при щелчке на кнопке в строке **type** (тип) в окне инспектора свойств. После выбора имени типа `Контроллер Банкомата` оно будет указано после двоеточия. Собственное имя линии жизни для удобства визуализации модели можно удалить. Аналогично может быть добавлена на диаграмму вторая линия жизни с именем типа `Устройство Чтения Карточки`. Диаграмма коммуникации после добавления второй линии жизни с открытым окном выбора типа будет иметь вид, представленный на рис. 14.29.

Для рассматриваемой модели системы управления банкоматом на диаграмму коммуникации следует добавить аналогичным образом линии жизни следующих типов: `Транзакция Банкомата`, `Контроллер Банка`, `Устройство Выдачи Наличных`, `Клавиатура`, `Принтер` и `Экран`.

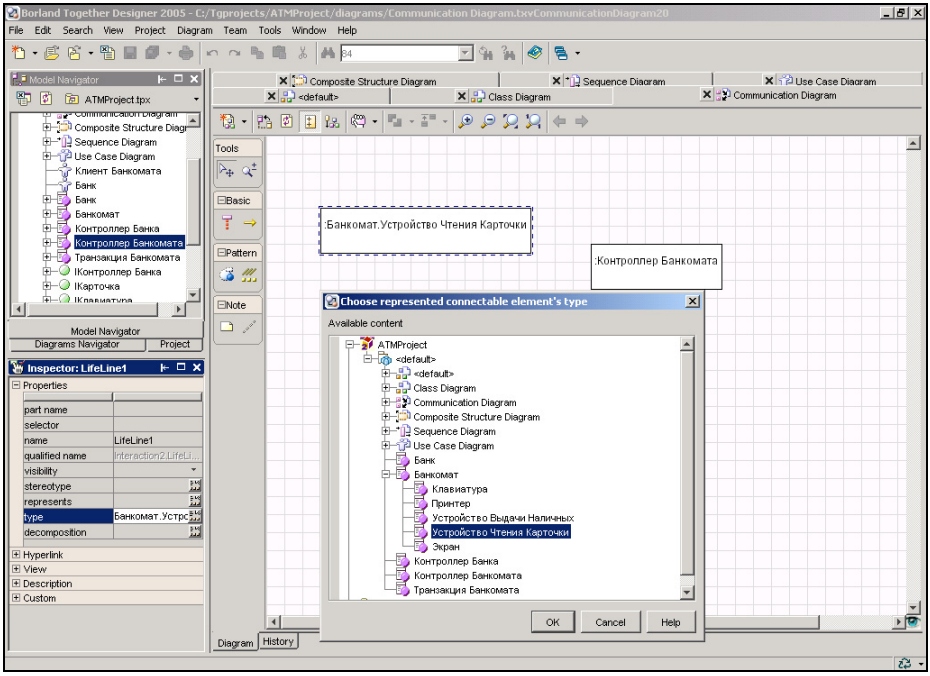


Рис. 14.29. Диаграмма коммуникации после добавления двух линий жизни

14.5.2. Добавление сообщений

Для добавления сообщения между предварительно размещенными на диаграмме линиями жизни нужно нажать кнопку с изображением сообщения на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении одной линии жизни на диаграмме и отпустить ее на изображении второй линии жизни. Имя сообщения выбирается из списка операций у класса, который является типом принимающей линии жизни.

В качестве первого сообщения добавим сообщение от линии жизни `Контроллер Банкомата` к линии жизни `Устройство Чтения Карточки`. Имя этого сообщения — "прочитать идентификатор карточки" — следует выбрать из списка операций класса `Устройство Чтения Карточки` в диалоговом окне, вызываемом при нажатии на кнопку в строке **signature** (сигнатура) в окне инспектора свойств этого сообщения. В результате выполнения этих действий на диаграмме коммуникации появится изображение сообщения с порядковым номером, задаваемым по умолчанию (рис. 14.30).

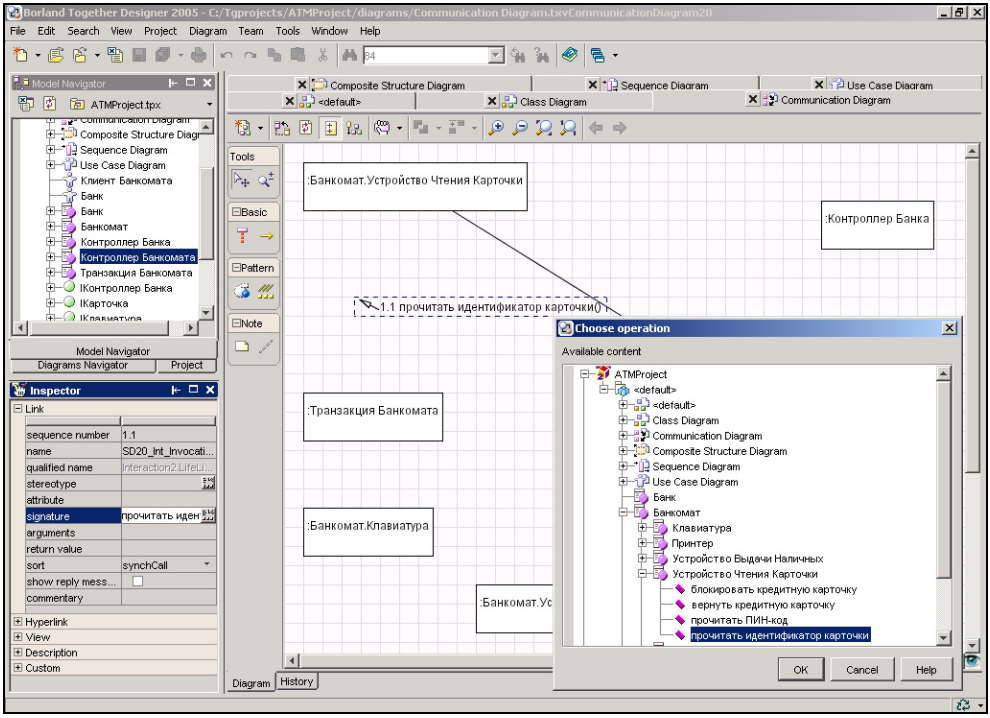


Рис. 14.30. Диаграмма коммуникации после добавления сообщения с дополнительным окном выбора операции

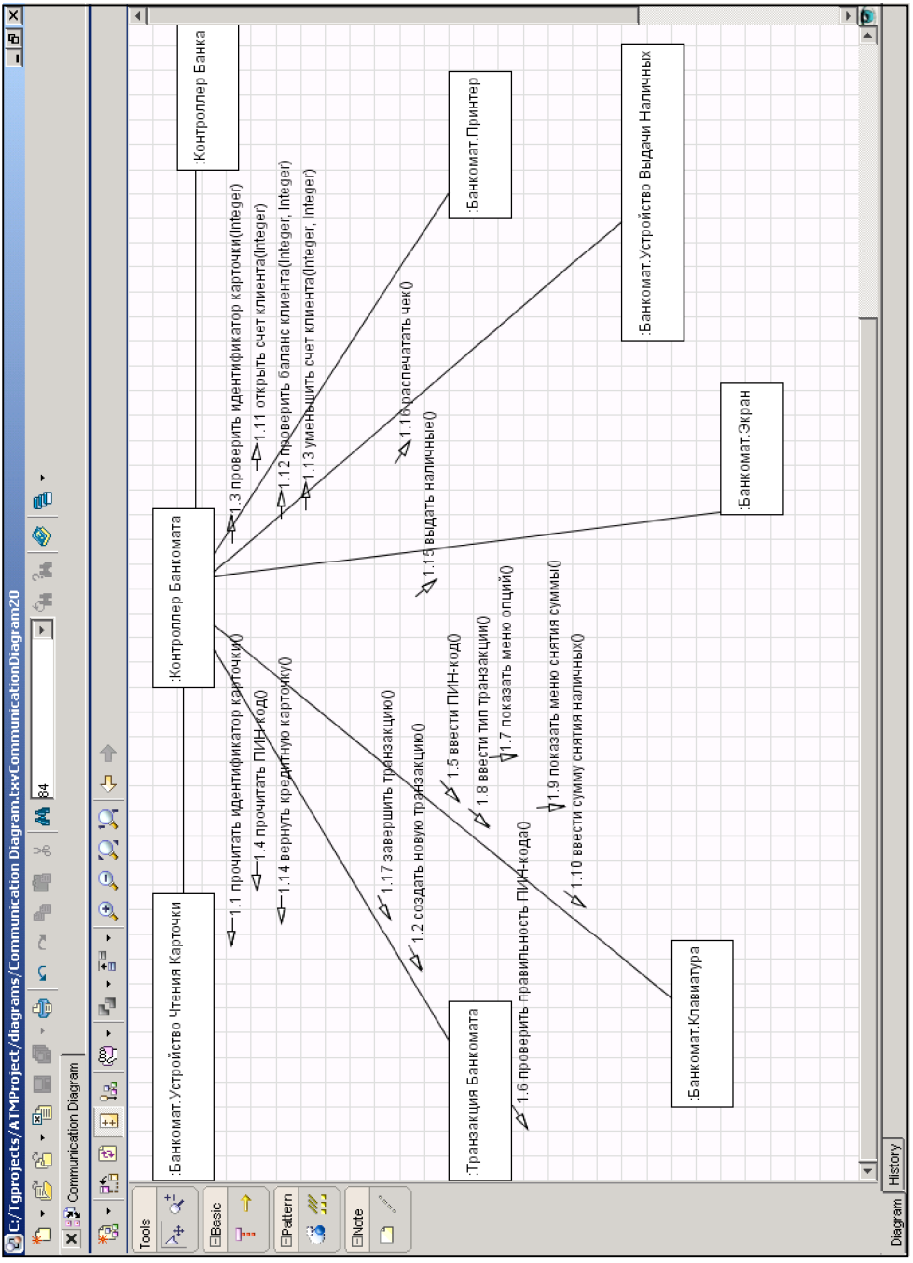


Рис. 14.31. Окончательный вид диаграммы коммуникации для варианта использования Снятие наличных по кредитной карточке

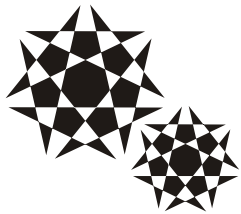
Следует заметить, что среди свойств сообщений в среде Together Designer отсутствует сторожевое условие. Этот факт может невольно огорчить разработчиков, поскольку затрудняет визуализацию альтернативности при реализации взаимодействия и приводит к увеличению количества диаграмм коммуникации в модели проекта. В качестве выхода из данной ситуации можно рекомендовать указать соответствующее сторожевое условие в качестве предусловия у соответствующей операции класса.

Для завершения построения диаграммы коммуникации рассматриваемого примера следует описанным выше способом добавить оставшиеся сообщения. Выполнить это предлагается читателям самостоятельно в качестве упражнения.

Диаграмма коммуникации, описывающая реализацию типичного хода событий варианта использования `Снятие наличных по кредитной карточке` системы управления банкоматом, будет иметь вид, как показано на рис. 14.31. Для удобства визуализации можно открыть ее в отдельном окне, для чего следует перетащить вкладку этой диаграммы из окна диаграммы на любое другое место рабочего интерфейса программы Together Designer.

Для завершения проекта системы управления банкоматом нужно разработать и другие типы канонических диаграмм, включая диаграммы физического представления. Из методических соображений разрабатываемая модель будет содержать все типы диаграмм, включая диаграммы конечного автомата, деятельности, компонентов и развертывания. Особенности построения этих диаграмм в среде Together Designer рассматриваются далее в *главе 15*.

Глава 15



Завершение разработки проекта в среде Borland Together Designer

В настоящей главе описывается процесс завершения разработки проекта в среде Together Designer. С этой целью рассматриваются особенности построения оставшихся типов канонических диаграмм. В заключение рассматриваются особенности генерации программного кода на языке Java в среде Together Designer для разработанной модели системы управления банкоматом.

Для продолжения работы над проектом системы управления банкоматом следует открыть разрабатываемый ранее проект. Это можно сделать, например, с помощью операции главного меню: **File | Open Project** (Файл | Открыть проект). В этом случае программа Borland Together загрузит проект ATMProject со всеми имеющимися в нем диаграммами, процесс создания которых был детально рассмотрен в *главе 14*.

15.1. Разработка диаграммы деятельности в среде Together Designer

Переходя к рассмотрению диаграммы деятельности, следует отметить, что в общем случае этот тип диаграмм может относиться к отдельному варианту использования, пакету, классу, операции класса или системе в целом. Нотация элементов диаграммы деятельности и общие рекомендации по ее построению были рассмотрены в *главе 8*.

Для разработки диаграммы деятельности рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены в *главе 14*. При выполнении действий по созданию новой диаграммы появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Activity**. При этом в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы деятельности.

Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы деятельности (табл. 15.1). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 15.1. *Назначение кнопок специальной панели инструментов диаграммы деятельности*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Activity	Добавляет на диаграмму деятельность
	Activity Parameter	Добавляет на диаграмму деятельности параметр деятельности
	Action	Добавляет на диаграмму действие
	Initial	Добавляет на диаграмму начальное состояние
	Activity Final	Добавляет на диаграмму конечное состояние
	Decision	Добавляет на диаграмму символ решения
	Merge	Добавляет на диаграмму символ слияния
	Fork	Добавляет на диаграмму символ разделения
	Join	Добавляет на диаграмму символ соединения
	Flow Final	Добавляет на диаграмму символ завершения потока
	Control Flow	Добавляет на диаграмму стрелку потока управления
	Input Pin	Добавляет на диаграмму входной контакт
	Output Pin	Добавляет на диаграмму выходной контакт
	Value Pin	Добавляет на диаграмму контакт значения
	Object node	Добавляет на диаграмму объект
	Central Buffer	Добавляет на диаграмму центральный буфер

Таблица 15.1 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Data Store	Добавляет на диаграмму хранилище данных
	Object flow	Добавляет на диаграмму стрелку потока объектов
	Accept Event Action	Добавляет на диаграмму действие ожидания события
	Accept Time Event Action	Добавляет на диаграмму действие ожидания временного события
	Send Signal Action	Добавляет на диаграмму действие передачи сигнала

Следует отметить, что достаточно большой набор элементов для диаграммы деятельности в языке UML 2.0 ориентирован на моделирование бизнес-процессов. Поскольку в нашем проекте модели бизнес-процессов не используются, эти элементы останутся невостребованными. Для разрабатываемого проекта системы управления банкоматом диаграмма деятельности будет описывать последовательность действий клиента и банкомата при реализации варианта использования `Снятие наличных по кредитной карточке`.

15.1.1. Добавление действий и деятельностей

Для добавления действия или деятельности на диаграмму деятельности нужно с помощью левой кнопки мыши нажать кнопку с изображением соответствующей пиктограммы на специальной панели инструментов, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение действия или деятельности с маркерами изменения его геометрических размеров и предложенным программой Together Designer именем по умолчанию, которое разработчику следует изменить. Сперва на диаграмму деятельности следует добавить начальное состояние и первое действие с именем `Вставить карточку` (рис. 15.1).

После добавления действия или деятельности на диаграмму деятельности можно выполнить редактирование их свойств в окне инспектора, в частности специфицировать локальные предусловия и постусловия.

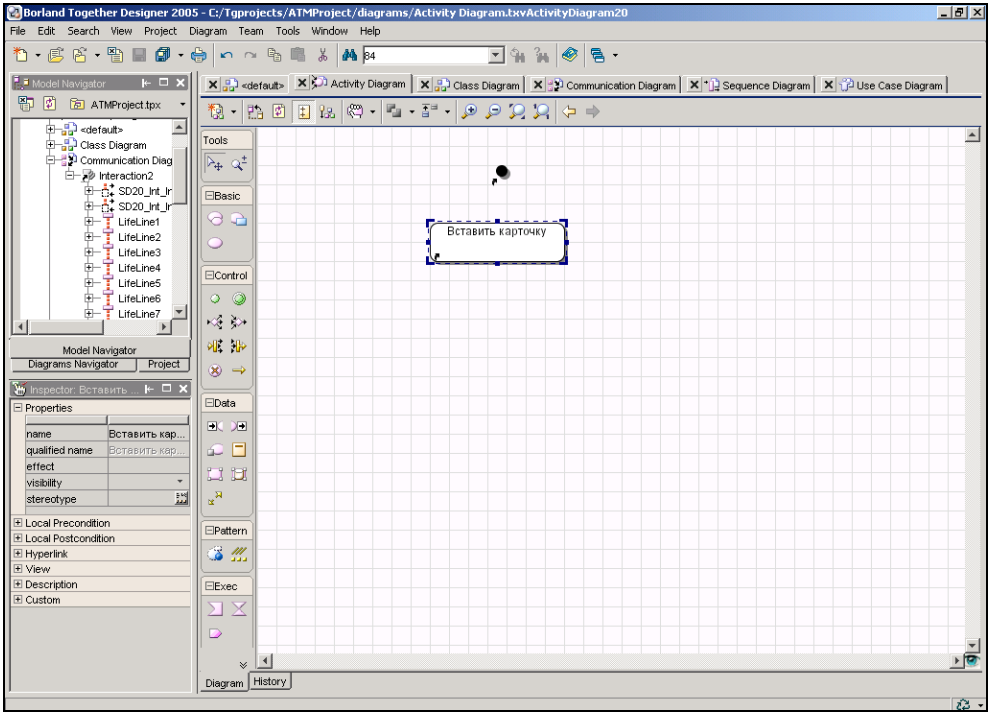


Рис. 15.1. Диаграмма деятельности после добавления на нее начального состояния и действия

15.1.2. Добавление потока управления

Напомним, что поток управления на диаграмме деятельности в нотации языка UML 2.0 обычно изображается горизонтально. Для добавления стрелки потока управления между двумя действиями или деятельностями нужно с помощью левой кнопки мыши нажать кнопку с изображением потока управления **Control Flow** на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении исходного элемента на диаграмме и отпустить ее на изображении целевого элемента. В результате этих действий на диаграмме появится изображение стрелки потока управления, соединяющей два выбранных элемента.

После добавления стрелки потока управления на диаграмму деятельности становятся доступными ее свойства в окне инспектора. При этом следует помнить, что все стрелки потока управления, выходящие из символов решения **Decision**, должны иметь сторожевые условия, которые специфицируются в строке **guard** окна инспектора свойств.

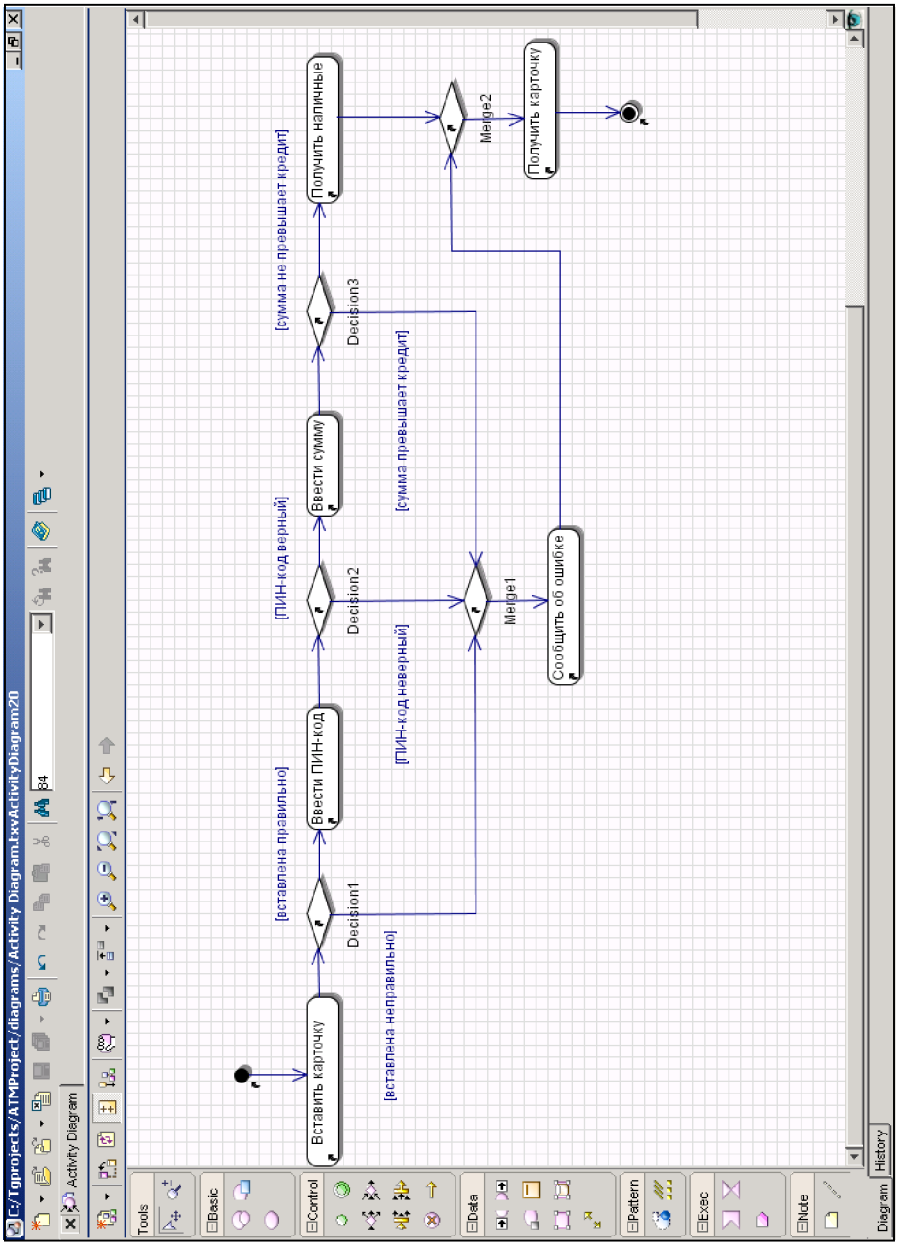


Рис. 15.2. Окончательный вид диаграммы деятельности разрабатываемой модели управления банкоматом

Для окончательного построения диаграммы деятельности рассматриваемого примера следует добавить остальные действия, финальное состояние, символы решения и ветвления, а также соответствующие стрелки потока управления. Построенная таким образом диаграмма деятельности будет иметь следующий вид (рис. 15.2).

На этом изображении для удобства позиционирования элементов на рабочем листе диаграммы уменьшен размер сетки. Это может быть выполнено на вкладке **Modeling** (Моделирование) окна свойств проекта, которое вызывается с помощью операции главного меню: **Project | Project Properties** (Проект | Свойства проекта).

Следует заметить, что в разрабатываемой модели диаграмма деятельности не является единственной, поскольку описывает лишь поведение системы управления банкоматом при реализации варианта использования `Снятие наличных по кредитной карточке`. Дополнить данную диаграмму информацией по реализации варианта использования `Получение справки о состоянии счета`, читателям предлагается самостоятельно в качестве упражнения.

Следует также отметить, что наличие в проекте двух диаграмм деятельности и конечного автомата часто является избыточным. Поэтому разработку диаграмм этого типа, особенно в условиях дефицита времени, отпущенного на выполнение проекта, иногда опускают. Однако в проектах реинжиниринга и документирования бизнес-процессов ситуация совершенно иная. В этих проектах диаграмма деятельности служит основным средством визуализации бизнес-процессов в контексте языка UML 2.0 и является обязательным артефактом этих проектов.

15.2. Разработка диаграммы конечного автомата в среде Together Designer

Диаграммы конечного автомата в общем случае могут относиться к отдельному пакету, классу или системе в целом. Нотация элементов диаграммы конечного автомата и общие рекомендации по ее построению были рассмотрены в *главе 10*.

Для разработки диаграммы конечного автомата рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены в *главе 14*. При выполнении действий по созданию новой диаграммы появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **State Machine**. При этом

в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы конечного автомата. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы конечного автомата (табл. 15.2). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 15.2. *Назначение кнопок специальной панели инструментов диаграммы конечного автомата*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	State	Добавляет на диаграмму состояние
	Entry Point	Добавляет на диаграмму точку входа
	Exit Point	Добавляет на диаграмму точку выхода
	Initial	Добавляет на диаграмму начальное состояние
	Final	Добавляет на диаграмму конечное состояние
	Terminate	Добавляет на диаграмму символ завершения
	Shallow History	Добавляет на диаграмму историческое состояние
	Deep History	Добавляет на диаграмму состояние глубокой истории
	Fork	Добавляет на диаграмму символ разделения
	Join	Добавляет на диаграмму символ слияния
	Choice	Добавляет на диаграмму символ принятия решения об альтернативном переходе
	Junction	Добавляет на диаграмму символ соединения
	Transition	Добавляет на диаграмму переход

Для разрабатываемого проекта системы управления банкоматом диаграмма конечного автомата будет построена для системы в целом и описывать ее поведение.

15.2.1. Добавление состояний

Для добавления состояния на диаграмму конечного автомата необходимо с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы состояния на специальной панели инструментов, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы.

На диаграмме появится изображение состояния с маркерами изменения его геометрических размеров и предложенным программой Together Designer именем по умолчанию, которое разработчику следует изменить. Вначале на диаграмму конечного автомата следует добавить начальное состояние и первое состояние с именем `Ожидание карточки` (рис. 15.3).

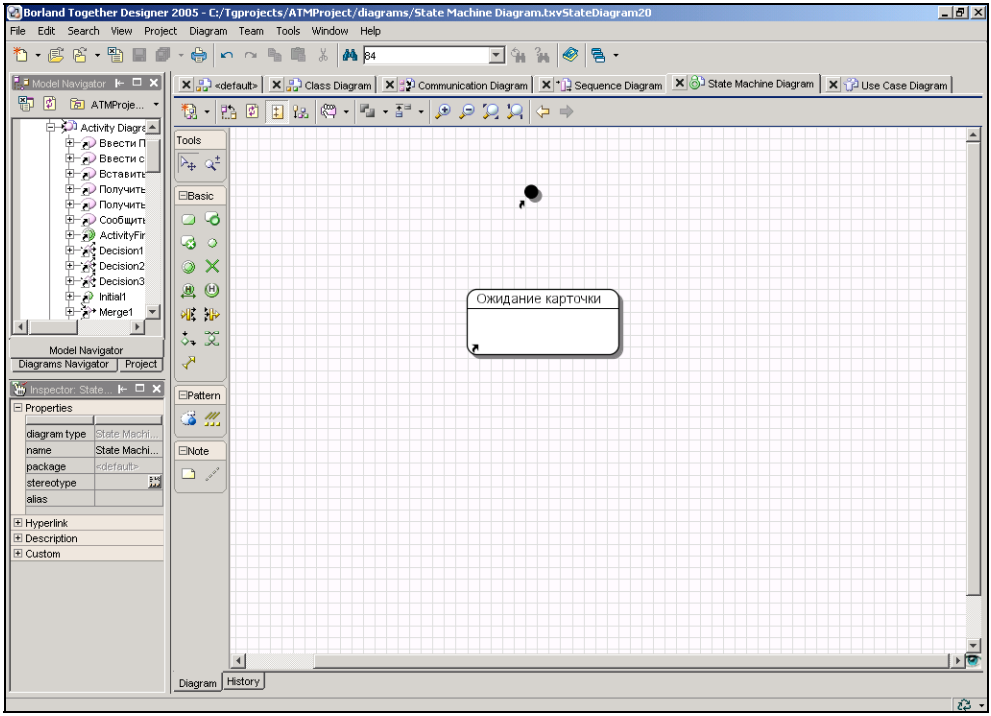


Рис. 15.3. Диаграмма конечного автомата после добавления на нее начального состояния и состояния `Ожидание карточки`

Редактирование свойств состояний выполняется в окне инспектора. При этом набор этих свойств в основном включает имя и стереотип состояния.

15.2.2. Добавление переходов

Добавление перехода на диаграмму конечного автомата выполняется аналогично стрелкам потока управления на диаграмме деятельности. А именно, для добавления перехода между двумя состояниями нужно с помощью левой кнопки мыши нажать кнопку с изображением перехода **Transition** на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении исходного состояния на диаграмме и отпустить ее на изображении целевого состояния. В результате этих действий на диаграмме появится изображение перехода, соединяющего два выбранных состояния.

После добавления перехода на диаграмму конечного автомата можно выполнить редактирование его свойств в окне инспектора. Следует обратить внимание, что имя триггерного перехода задается в строке **trigger**, а условие на переходе — в строке **guard**. Эта строка служит для задания сторожевого условия, которое определяет возможность срабатывания перехода. В строке **effect** следует специфицировать действие, которое происходит при срабатывании перехода.

Для окончательного построения диаграммы конечного автомата рассматриваемого примера следует аналогичным образом добавить остальные состояния и переходы. Построенная таким образом диаграмма конечного автомата будет иметь вид, представленный на рис. 15.4. На этой диаграмме также изображены точки выхода из двух состояний, которые моделируют возможность прекращения выполнения транзакции клиентом.

При необходимости можно декомпозировать состояния `Обработка запроса на снятие наличных` и `Обработка запроса на получение справки`, изобразив их внутреннюю структуру в виде вложенных подсостояний. Выполнить эти действия предлагается читателям самостоятельно в качестве упражнения.

Следует заметить, что в разрабатываемой модели диаграмма конечного автомата является единственной и описывает поведение системы управления банкоматом в целом. Главное достоинство данной диаграммы — возможность визуализировать условный характер поведения встроенных систем или систем реального времени, к которым относится и банкомат. Для других типов программных систем, например, систем управления базами данных или интернет-приложений, разработка диаграммы конечного автомата не является обязательной. Как правило, для таких систем оказывается достаточно одной из двух диаграмм взаимодействия — диаграммы последовательности или кооперация.

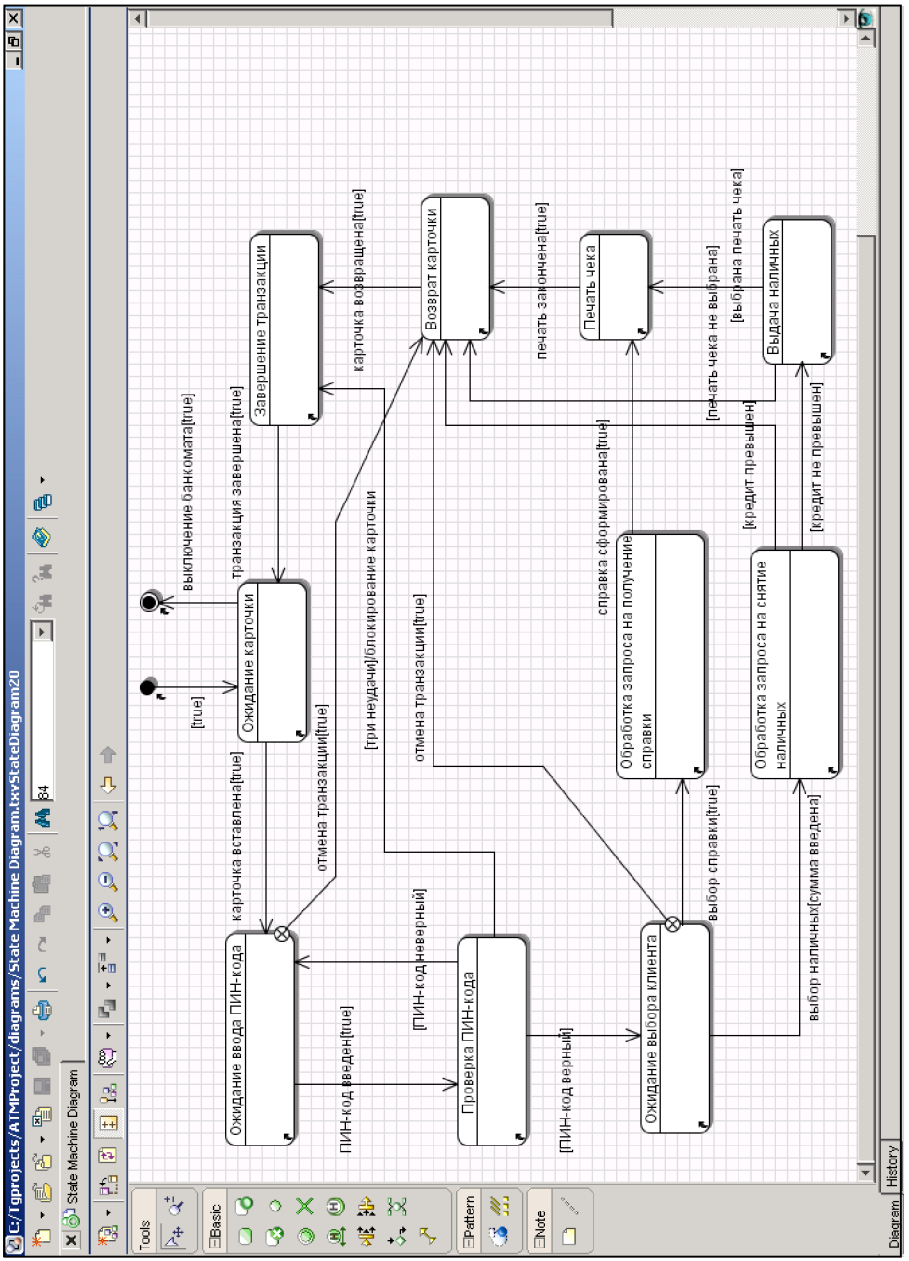


Рис. 15.4. Окончательный вид диаграммы конечного автомата разрабатываемой модели управления банкоматом

15.3. Разработка диаграммы компонентов в среде Together Designer

Диаграмма компонентов служит частью физического представления модели и занимает важное место среди диаграмм проекта. Нотация элементов диаграммы компонентов и общие рекомендации по ее построению были рассмотрены в *главе 11*.

Для разработки диаграммы компонентов рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены в *главе 14*. При выполнении действий по созданию новой диаграммы появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Component**. При этом в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы компонентов. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы компонентов (табл. 15.3). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 15.3. *Назначение кнопок специальной панели инструментов диаграммы компонентов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Component	Добавляет на диаграмму компонент
	Port	Добавляет на диаграмму порт
	Artifact	Добавляет на диаграмму артефакт
	Class	Добавляет на диаграмму класс
	Interface	Добавляет на диаграмму интерфейс
	Instance Specification	Добавляет на диаграмму спецификацию экземпляра или объекта
	Delegation Connector	Добавляет на диаграмму соединитель
	Provided Interface	Добавляет на диаграмму предоставляемый интерфейс

Таблица 15.3 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Required Interface	Добавляет на диаграмму требуемый интерфейс
	Association	Добавляет на диаграмму отношение ассоциации
	Aggregation	Добавляет на диаграмму отношение агрегации
	Dependency	Добавляет на диаграмму отношение зависимости
	Realization	Добавляет на диаграмму отношение реализации

15.3.1. Добавление компонентов

Добавление и удаление компонентов на диаграмму компонентов выполняется стандартным образом с помощью кнопок с изображением пиктограммы требуемого компонента. После добавления компонента на диаграмму следует определить его имя. С помощью соответствующих маркеров можно изменить также геометрические размеры компонента. Добавим на диаграмму компонентов два компонента с именами: `Главная программа` и `Транзакция` (рис. 15.5). Дополнительно в качестве стереотипа для первого из них выберем из вложенного списка в строке свойства **stereotype** значение `process` (процесс), а для второго — значение `entity` (сущность).

Кроме этих двух компонентов следует добавить также компоненты `Устройства` и `Программа обслуживания банкоматов`, а также скопировать на диаграмму из навигатора модели интерфейс `IКонтроллер Банка`. После этого можно приступить к спецификации классов, которые реализуют эти компоненты.

С этой целью необходимо выполнить следующие действия:

1. Скопировать класс `Контроллер Банкомата` из навигатора модели в компонент `Главная программа`.
2. Скопировать класс `Транзакция Банкомата` из навигатора модели в компонент `Транзакция`.
3. Скопировать класс `Контроллер Банка` из навигатора модели в компонент `Программа обслуживания банкоматов`.
4. Скопировать классы `Устройство Чтения Карточки`, `Устройство Выдачи Наличных`, `Клавиатура`, `Экран` и `Принтер` из навигатора модели в компонент `Устройства`.

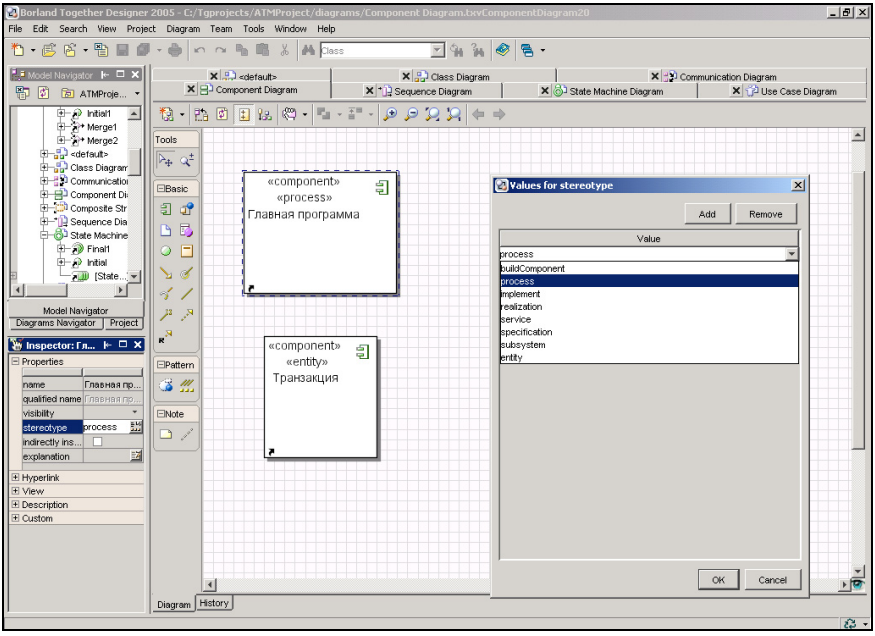


Рис. 15.5. Диаграмма компонентов после добавления компонентов `Главная программа` и `Транзакция` с окном выбора стереотипа

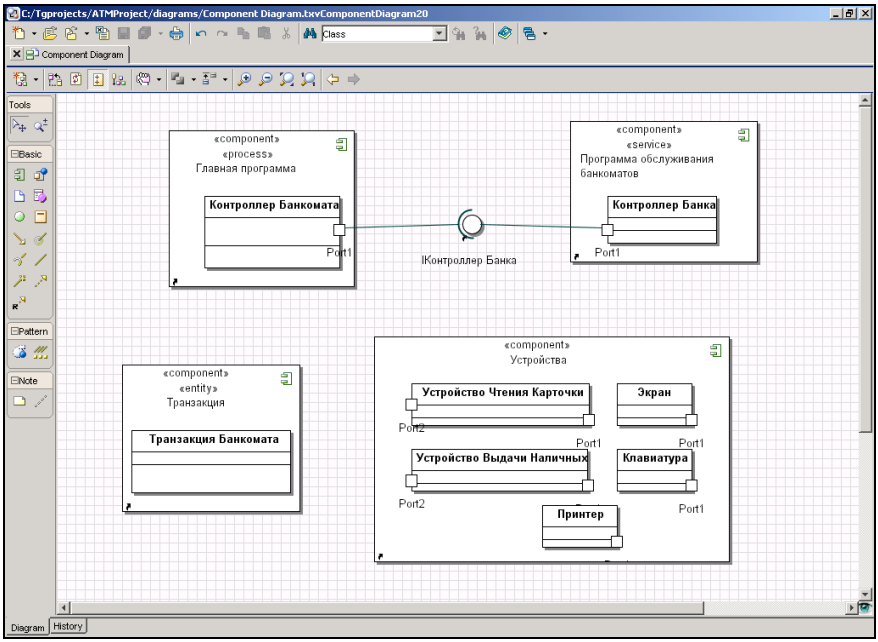


Рис. 15.6. Диаграмма компонентов после добавления классов к компонентам

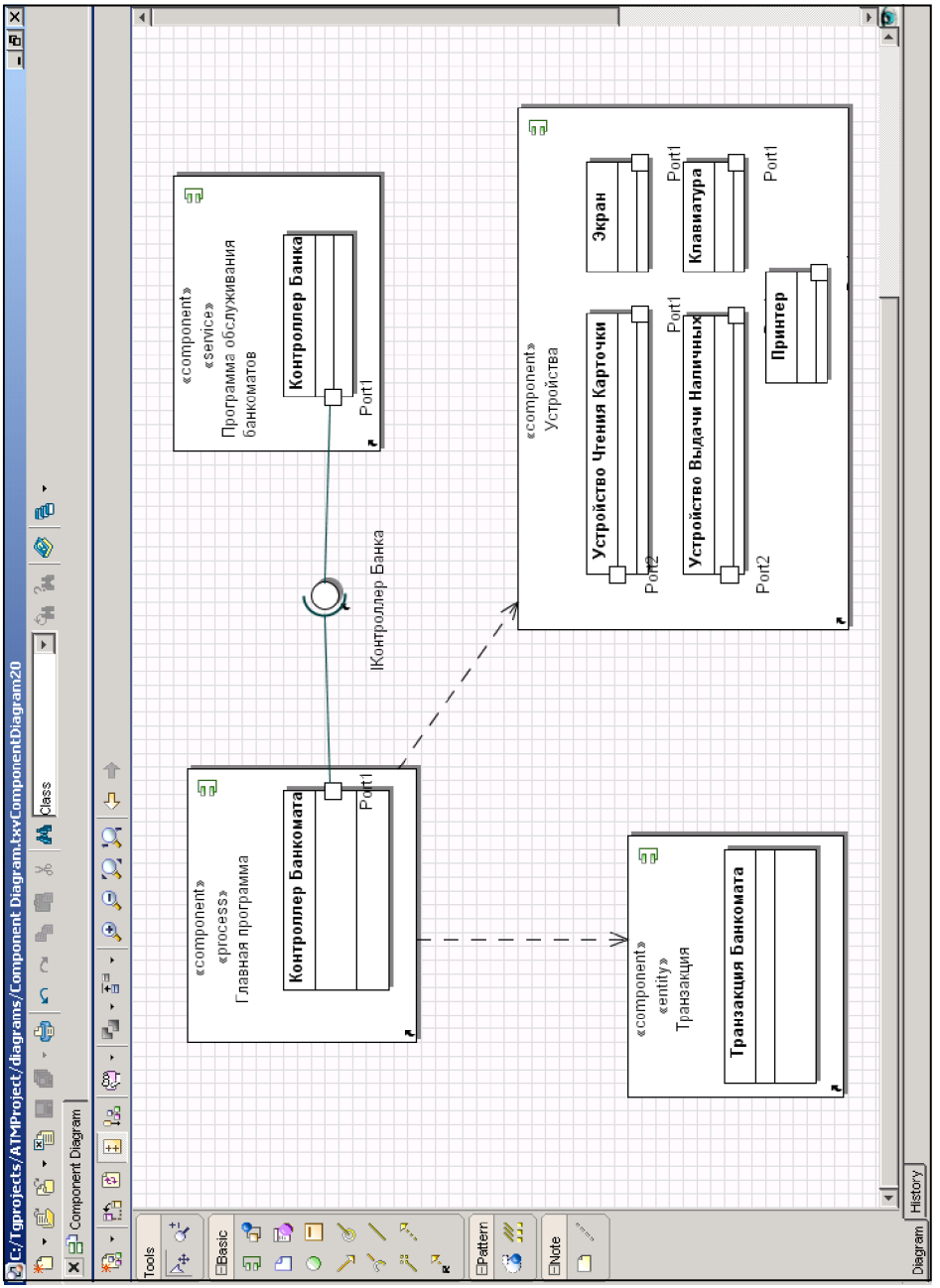


Рис. 15.7. Окончательный вид диаграммы компонентов разрабатываемой модели управления банкоматом

Появившиеся при копировании ассоциации между классами желательно сделать невидимыми на диаграмме, для чего следует выполнить операцию **Hide** (Скрыть) в контекстном меню соответствующей ассоциации. Скрыть атрибуты и операции классов можно нажав на знак "–", расположенный в левой части соответствующей секции класса. Построенная таким образом диаграмма компонентов будет иметь следующий вид (рис. 15.6).

Следует отметить, что связи интерфейса с соответствующими классами появляются автоматически после добавления этих классов к компонентам. При необходимости можно скрыть и порты устройств банкомата с помощью указанной операции контекстного меню.

15.3.2. Добавление отношений на диаграмму компонентов

Хотя на диаграмму компонентов можно добавить такие же отношения, что и на диаграмму классов, наиболее часто из этих отношений используется зависимость между компонентами. При этом добавление отношения зависимости на диаграмму компонентов аналогично добавлению отношений на других ранее рассмотренных диаграммах. В результате выполнения соответствующих действий на диаграмме компонентов появится изображение отношения зависимости между двумя компонентами.

Для окончательного построения диаграммы компонентов рассматриваемого примера следует добавить отношения зависимости между компонентами. Построенная таким образом диаграмма компонентов будет иметь вид, представленный на рис. 15.7.

При работе с диаграммой компонентов можно создавать компоненты и размещать в них классы, добавлять к компонентам порты и спецификации экземпляров, задавать специальные стереотипы для отдельных разновидностей компонентов, указывать различные типы отношений между компонентами и классами. Выполнить эти действия предлагается читателям самостоятельно в качестве упражнения.

15.4. Разработка диаграммы развертывания в среде Together Designer

Диаграмма развертывания является второй моделью физического представления системы и разрабатывается, как правило, для территориально распределенных программных систем. Нотация элементов диаграммы развертывания и общие рекомендации по ее построению были рассмотрены в *главе 12*.

Для разработки диаграммы развертывания рассматриваемой модели банкомата необходимо создать и активизировать ее в окне диаграммы. Это можно сделать различными способами, аналогично тем, которые были рассмотрены в *главе 14*. При выполнении действий по созданию новой диаграммы появится окно с галереей объектов для выбора типа новой диаграммы (см. рис. 14.3), в котором следует выбрать вариант **Deployment**. При этом в окне диаграммы появится новая вкладка с чистым рабочим листом диаграммы развертывания. Специальная панель инструментов будет содержать кнопки с изображением пиктограмм элементов, необходимых для разработки диаграммы развертывания (табл. 15.4). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 15.4. *Назначение кнопок специальной панели инструментов диаграммы развертывания*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Node	Добавляет на диаграмму узел
	Device	Добавляет на диаграмму устройство
	Execution environment	Добавляет на диаграмму среду выполнения
	Artifact	Добавляет на диаграмму артефакт
	Deployment Specification	Добавляет на диаграмму спецификацию развертывания
	Deployment	Добавляет на диаграмму отношение развертывания
	Generalization	Добавляет на диаграмму отношение обобщения
	Association	Добавляет на диаграмму отношение ассоциации
	Dependency	Добавляет на диаграмму отношение зависимости
	Manifestation	Добавляет на диаграмму отношение манифестации
	Communication path	Добавляет на диаграмму путь коммуникации

15.4.1. Добавление узлов, сред выполнения и компонентов

Добавление узла на диаграмму развертывания выполняется стандартным образом с помощью кнопок с изображением пиктограммы узла **Node**. После добавления узла на диаграмму следует определить его имя, а с помощью соответствующих маркеров можно изменить его геометрические размеры. Добавим на диаграмму развертывания два узла с именами `Банкомат` и `Банк` (рис. 15.8).

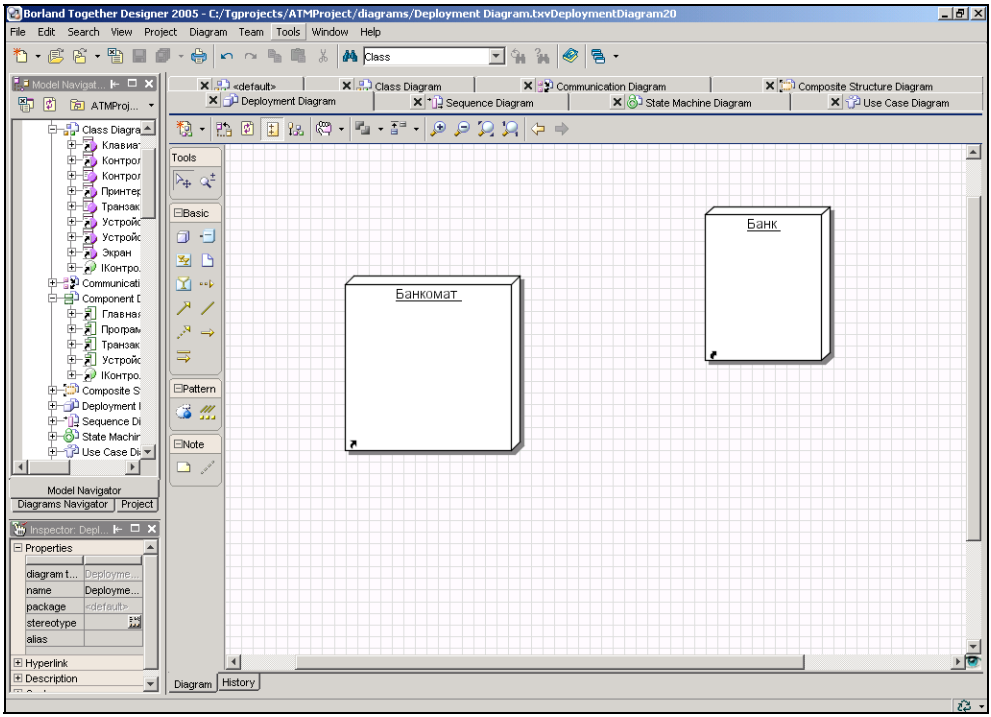


Рис. 15.8. Диаграмма развертывания после добавления двух узлов

Для каждого узла дополнительно можно определить стереотип и псевдоним (alias). Спецификация этих свойств выполняется в окне инспектора. Далее следует выполнить следующие действия:

1. Добавить на узел `Банкомат` среду выполнения **Execution environment** с именем J2ME (Java 2 Micro Edition).
2. Добавить на узел `Банк` среду выполнения **Execution environment** с именем J2EE (Java 2 Enterprise Edition).

3. Добавить в среду выполнения J2ME три артефакта с именами: `Главная программа`, `Транзакция` и `Устройства`.
4. Добавить в среду выполнения J2EE артефакт с именем `Программа обслуживания банкоматов`.

Построенная таким образом диаграмма компонентов будет иметь вид, представленный на рис. 15.9.

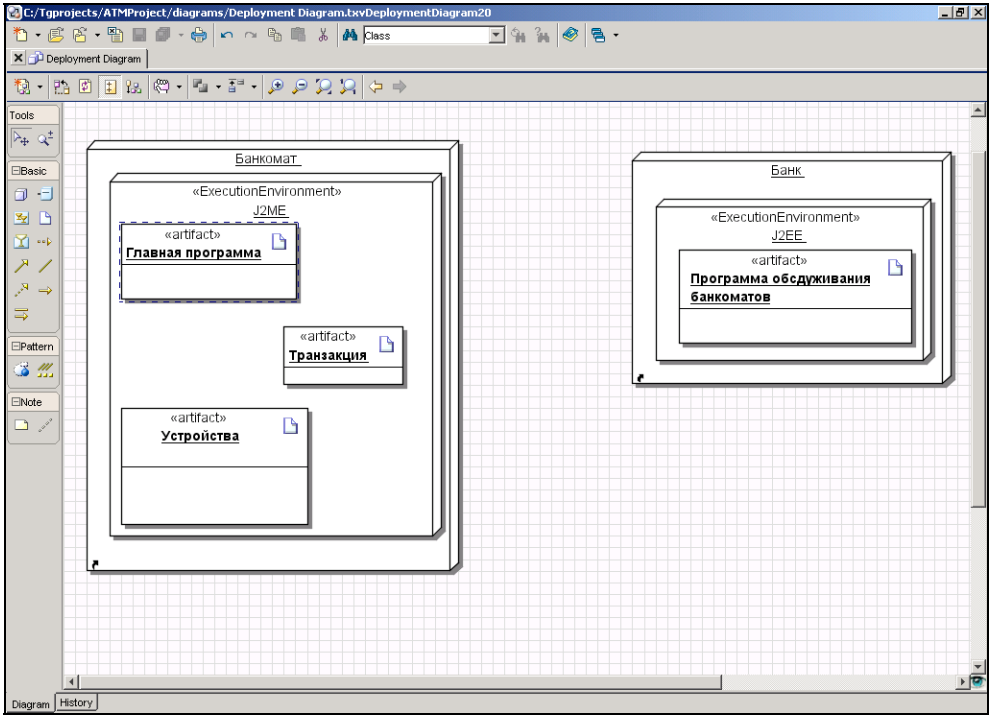


Рис. 15.9. Диаграмма развертывания
после добавления сред выполнения и артефактов

15.4.2. Добавление отношений на диаграмму развертывания

Добавление отношений на диаграмму развертывания осуществляется аналогично ранее рассмотренным способам добавления отношений на других диаграммах. После выбора и нажатия нужной кнопки с отношением на специальной панели инструментов на диаграмме развертывания появится изображение соответствующей линии между двумя элементами.

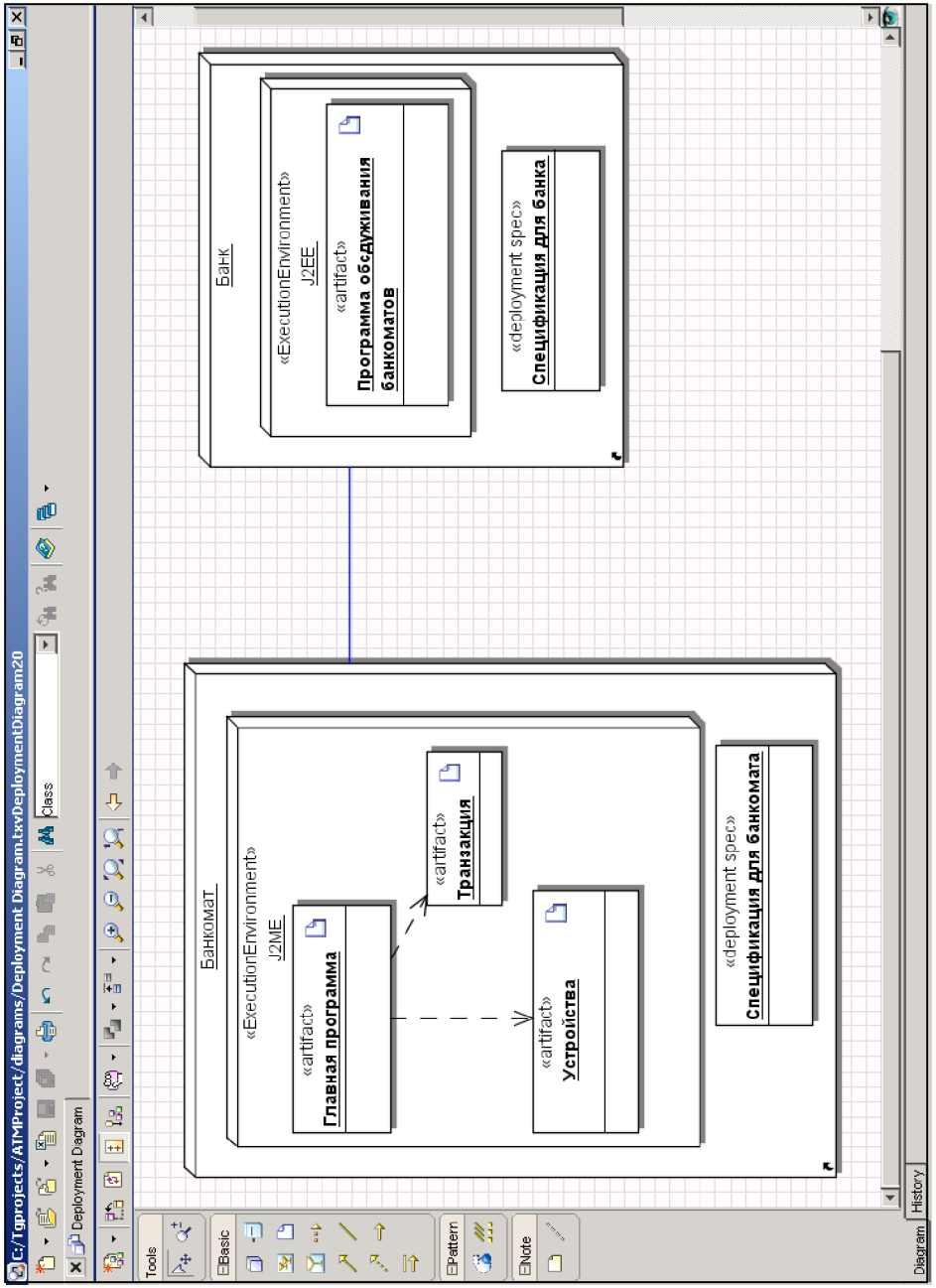


Рис. 15.10. Окончательный вид диаграммы развертывания разрабатываемой модели управления банкоматом

Для окончательного построения диаграммы развертывания рассматриваемого примера добавим две спецификации развертывания на узлы `Банкомат` и `Банк`, зависимость от артефакта `Главная программа` к артефактам `Транзакция` и `Устройства`, а также путь коммуникации между узлами `Банкомат` и `Банк`. Построенная таким образом диаграмма развертывания будет иметь вид, представленный на рис. 15.10.

Построением диаграммы развертывания завершается разработка модели в нотации языка UML 2.0 для системы управления банкоматом.

15.5. Генерация документации и программного кода в среде Together Designer

Дальнейшая работа с разработанной моделью зависит от целей выполнения проекта. Если проект не предполагает программной реализации, то можно ограничиться формированием проектной документации. С этой целью следует выполнить операцию главного меню **Project | Documentation** (Проект | Документация), в результате чего будет открыто диалоговое окно настройки свойств генерации отчета. Нажатие кнопки **OK** в этом окне приведет к автоматической генерации отчета о разрабатываемой модели в формате HTML. Для разработанной модели банкомата окно с отчетом появится сразу после завершения его формирования в качестве отдельной вкладки в окне диаграмм (рис. 15.11).

Если проект предполагает программную реализацию, то можно воспользоваться возможностью среды Together Designer по автоматической генерации программного кода на языке Java на основе разработанной модели. Программа генерирует текст программного кода на языке Java для всех классов модели, которые расположены на диаграмме классов по умолчанию открытого проекта или в узле **<default>** навигатора модели. Предварительного построения диаграммы компонентов для этого не требуется. Предварительно выделять эти классы или открывать диаграмму классов также не обязательно. Все, что необходимо сделать для генерации кода, — это выполнить операцию главного меню **Tools | Generate Java Source Code** (Инструменты | Генерировать код на языке Java).

В результате выполнения этой операции будет открыто диалоговое окно, в котором необходимо выбрать папку для размещения файлов с текстом программного кода на языке программирования Java. После выбора соответствующей папки и нажатия кнопки **OK** программа Together Designer выполнит

генерацию кода. По завершении генерации программного кода в выбранной папке появится более десятка файлов с текстом кода на языке Java. При этом также отобразится окно сообщений с предупреждениями о том, что программа обнаружила недопустимые пробелы в объявлении классов, атрибутов и операций и заменила пробелы символом связного подчеркивания.

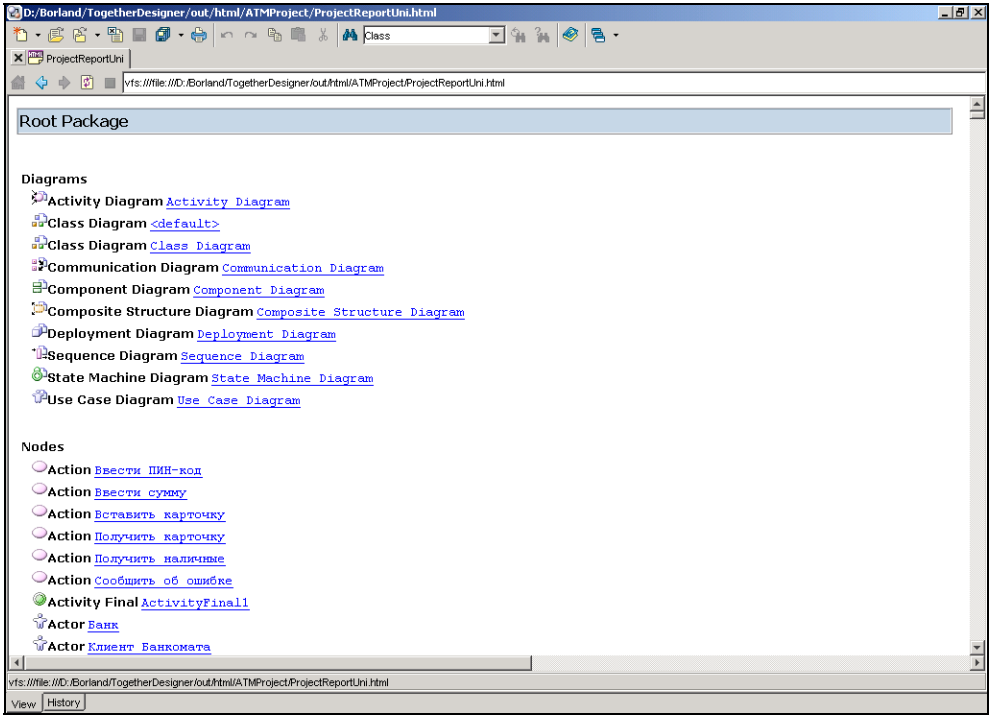


Рис. 15.11. Окно с отчетом для разработанной модели банкомата

Для просмотра и редактирования этих файлов можно воспользоваться обычным текстовым редактором Блокнот. В качестве примера рассмотрим содержание файла Банкомат.java (рис. 15.12).

Как видно из рассмотрения этого файла, в нем содержится объявление, в соответствии с правилами синтаксиса языка Java, всех классов, входящих в состав композитного класса `Банкомат`, а также атрибутов и операций этих классов. При этом каждая из операций имеет пустое тело реализации, которое должен написать программист самостоятельно, исходя из функциональных требований модели и синтаксиса языка программирования Java. Данную работу удобнее выполнять в среде программирования, например, Borland

JBuilder. При использовании среды программирования, кроме компиляции, отладки и тестирования исходных модулей программы разработчик получает возможность дополнить приложение графическим интерфейсом, необходимым для взаимодействия с пользователем.

Банкомат.java - Блокнот

Файл Правка Формат Справка

```
public class Банкомат {
   private class Клавиатура {
      /**
       * Вызывается после окончания ввода значения снимаемой суммы с клавиатуры
       */
      public int ввести_сумму_снятия_наличных(){}
      /**
       * Возвращает значение Истина, если клиент выбирает снятие наличных, и значение Ложь, если
клиент выбирает получение справки о состоянии счета
       */
      public boolean ввести_тип_транзакции(){}
      /**
       * Вызывается после окончания ввода значения ПИН-кода с клавиатуры
       */
      public int operation1(){}
      private int lnkAttribute1;
   }
   private class Принтер {
      /**
       * Вызывается по дополнительному запросу клиента
       */
      public void распечатать_чек(){}
      private int lnkAttribute1;
   }
   private class Устройство_Выдачи_Наличных {
      /**
       * Вызывается после того, как выполнено снятие запрошенной клиентом суммы со счета
       */
      public void выдать_наличные(){}
   }
   private class Устройство_Чтения_Карточки {
      /**
       * Вызывается после того, как трижды введен неправильный ПИН-код или установлен факт
утраты кредитной карточки владельцем
       */
      public void блокировать_кредитную_карточку(){}
      /**
```

Рис. 15.12. Текстовый редактор Блокнот с загруженным в него файлом Банкомат.java

Следует заметить, что программа Together Designer по замыслу своих разработчиков предназначена только для дизайна моделей, т. е. возможность кодогенерации в этой редакции хотя и предусмотрена, но, мягко говоря, оставляет желать лучшего. Наибольшее впечатление производит возможность синхронизации кода и моделей, разработанных в нотации UML 1.*x*/2.0, при

интеграции этой программы с инструментальными средами программирования, такими как Borland® Together® Developer for JBuilder® и Borland® Together® Developer for MS .NET®. И, наконец, нельзя не упомянуть о среде Borland® Delphi® Studio, в которой в полном объеме реализованы возможности разработки приложений, управляемых моделями в нотации UML 1.*x*/2.0.

Все вопросы по разработке диаграмм в нотации языка UML 2.0, которые были рассмотрены в книге применительно к программе Together Designer, остаются справедливыми для любой другой редакции этой программы. Изучив материал книги, читатели без особого труда смогут освоить редактор моделей в Borland® Delphi® Studio и самостоятельно разрабатывать приложения, управляемые моделями. Более того, даже при появлении новых версий программы Together Designer, изложенные базовые приемы и операции по разработке моделей вряд ли претерпят серьезные изменения.

В заключение следует отметить, что наибольший эффект от использования Together Designer проявляется при разработке масштабных проектов. Действительно, при рассмотрении простых моделей может сложиться впечатление о том, что написать и отладить соответствующую программу гораздо проще непосредственно в той или иной интегрированной среде программирования. Однако ситуация оказывается не столь тривиальной, когда необходимо выполнить проект с сотней классов, каждый из которых содержит более десятка операций. Именно при выполнении подобных проектов явно выявляется преимущество использования средства Together Designer и нотации языка UML 2.0 для документирования и реализации соответствующей модели.

Заключение

В настоящее время полностью специфицирована и документирована версия 2.0 языка UML и продолжается дальнейшая работа по его развитию. Ход этой работы и ее состояние отражаются на официальном сайте консорциума OMG **www.omg.org**. Там же содержатся полные спецификации стандарта OMG-UML, предоставленные для свободного доступа. Другим источником информации по разработке моделей в нотации UML является сайт компании IBM Corp. **www.ibm.com**, которая, после приобретения Rational Software, продолжает разработку CASE-средств, поддерживающих нотацию языка UML 2.0.

В последнее время появилось множество программных средств, позволяющих преобразовывать форматы представления графических диаграмм языка UML. Так, например, на сайте **www.reischmann.com** содержится информация о нескольких десятках подобных программ, реализующих преобразование графических моделей в нотации языка UML между популярными CASE-средствами.

Из отечественных ресурсов следует отметить сайт компании "Интерфейс" — **www.interface.ru**, где содержится информация по многим современным CASE-средствам, рассматриваются их характеристики и возможности, а также предлагаются учебные курсы по обучению нотации языка UML 2.0 и работе с соответствующими CASE-средствами.

Перспективы дальнейшего развития UML связаны со становлением и интенсивным развитием новой парадигмы объектно-ориентированного анализа и проектирования — сервисно-ориентированной архитектуры (Service-Oriented Architecture, SOA). В этой связи развернута работа над дополнительной спецификацией языка UML применительно к сервисам .NET и SOA. Речь идет о разработке так называемых *профилей*, содержащих нотацию всех необходимых элементов для представления в языке UML 2.0 компонентов соответствующих технологий. При этом интенсивно используется механизм расширения языка UML 2.0 за счет добавления новых стереотипов, помеченных значений и ограничений.

Язык UML уже сейчас находит широкое применение в качестве неофициального стандарта в процессе разработки программных систем, связанных с такими об-

ластями, как моделирование бизнеса, управление требованиями, анализ и проектирование, программирование и тестирование. Применительно к этим процессам в языке UML унифицированы стандартные обозначения основных элементов соответствующих предметных областей.

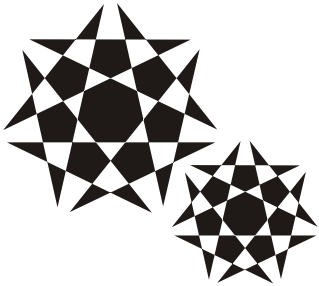
Следует также отметить, что развитие языка UML 2.0 на основе включения в его нотацию дополнительных элементов и стереотипов стимулирует разработку соответствующих инструментальных CASE-средств. Можно с уверенностью предположить, что эта область развития информационных технологий имеет широчайшие перспективы и стратегическое значение не только в качестве языка общения между заказчиками и разработчиками программных систем, но и для документирования проектов в целом. При этом достигается требуемый уровень стандартизации и унификации всех используемых для этой цели обозначений.

Разработав модель и специфицировав ее на языке UML, разработчик имеет все основания быть понятым и по достоинству оцененным своими коллегами. При этом невозможны ситуации, когда тот или иной разработчик применяет собственную графическую нотацию для представления тех или иных аспектов модели, что практически исключает ее адекватное понимание другими специалистами.

Не менее важный аспект применения языка UML связан с профессиональной подготовкой соответствующих специалистов. Речь идет о том, что знания различных научных дисциплин характеризуют различные аспекты реального мира. При этом принципы системного анализа позволяют рассматривать те или иные объекты в качестве систем.

Язык UML 2.0 не имеет аналогов в качестве "эсперанто", на котором могут общаться между собой руководители информационных служб, менеджеры проектов, системные аналитики, бизнес-аналитики, программисты, тестировщики, математики, экономисты, физики и специалисты других профессий, представляя свои профессиональные знания в унифицированном виде. Ведь, по существу, каждый из специалистов оперирует модельными представлениями в своей области знаний. И именно этот модельный аспект может быть специфицирован и документирован средствами языка UML.

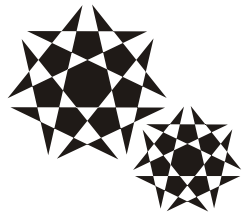
В связи с этим значение языка UML 2.0 неуклонно возрастает, поскольку он все более приобретает черты языка представления знаний. При этом наличие в языке UML 2.0 изобразительных средств для представления структуры и поведения модели позволяет достичь адекватного представления декларативных и процедурных знаний и, что не менее важно — установить между этими формами знаний семантическое соответствие. Все эти особенности языка UML 2.0 позволяют сделать вывод о том, что он занимает лидирующее положение среди всех графических нотаций моделирования и документирования программных систем.



ПРИЛОЖЕНИЯ

Приложение 1. Язык объектных ограничений OCL

Приложение 2. Глоссарий



Приложение 1

Язык объектных ограничений OCL

Язык объектных ограничений (Object Constraint Language, OCL) представляет собой формальный язык, предназначенный для описания ограничений, которые могут быть применены к любым элементам языка UML 2.0. Хотя по определению язык OCL является частью языка UML 2.0, в общем случае он имеет более широкую область приложений. Средства языка OCL позволяют специфицировать не только объектные ограничения, но и другие выражения логико-лингвистического характера, такие как предусловия, постусловия и сторожевые условия. Все это делает его чрезвычайно мощным инструментом системного моделирования.

В настоящее время язык OCL поддерживается во многих инструментальных средах разработки, включая Borland® Together® Designer и Borland® Delphi® Studio. В этих средах имеется специальный текстовый редактор выражений OCL, в котором выполняется проверка их синтаксиса и выделение цветом выражений, содержащих ошибки.

При разработке моделей в нотации UML 2.0 чисто графических средств может оказаться недостаточно для спецификации точных и непротиворечивых моделей сложных систем. Может возникнуть необходимость задания дополнительных ограничений, которым должны удовлетворять различные элементы модели. Традиционно подобные ограничения описывались с помощью естественного языка или псевдокода. Однако, как показала практика системного и визуального моделирования в контексте ООАП, сформулированные на естественном языке ограничения страдают неоднозначностью и нечеткостью.

Для преодоления недостатков естественного языка были разработаны различные формальные языки, которые придают рассуждениям строгий характер. Однако эти формальные языки требуют для своего конструктивного использования знания основ математической логики и теории формального

вывода. С одной стороны, это делает формальные языки естественным средством для построения математических моделей. С другой стороны, существенно ограничивает круг потенциальных пользователей, поскольку разработка формальных моделей требует специальной квалификации.

Другая особенность традиционных формальных языков заключается в присущей им "бедной" семантике. Речь идет о том, что базовые элементы формальных языков имеют слишком абстрактное содержание, что затрудняет их непосредственную интерпретацию в понятиях моделей конкретных технических систем и бизнес-процессов. Поскольку процесс ООАП направлен на построение конструктивных моделей сложных систем, которые должны быть реализованы в форме программных систем и аппаратных комплексов, это является серьезным недостатком. Речь идет о необходимости применения такого формального языка, базовые элементы которого адекватно отражают семантику основных конструкций ООАП.

Именно для этих целей и был разработан язык OCL. По своей сути он также является формальным языком, однако более простым для изучения, чем традиционные формально-логические языки. В то же время язык OCL специально ориентирован на описание бизнес-процессов и бизнес-логики, поскольку был разработан в одном из отделений корпорации IBM Джосом Уормером (Jos Warmer). В историческом контексте язык OCL представляет собой дальнейшее развитие метода Syntropy Стива Кука (Steve Cook) и Джона Дэниэлса (John Daniels).

Язык OCL является языком формальных выражений, которые используются для записи отдельных ограничений. Этот язык ничего не изменяет в графических моделях, а только дополняет их. Выражения OCL не могут расширить или изменить семантику существующих элементов языка UML 2.0, они могут только сузить ее.

Выражения OCL не имеют побочных эффектов. Это означает, что выражения языка OCL никогда не могут изменить состояние системы, хотя эти выражения и могут быть использованы для спецификации самого процесса изменения этого состояния. Выражения языка OCL также не могут изменять отдельные значения атрибутов и операций для объектов и их связей. Всякий раз, когда оценивается некоторое такое выражение, на выходе получается лишь некоторое значение.

Язык OCL не является языком программирования в обычном смысле, поскольку не позволяет записать логику выполнения программы или последовательность управляющих действий. Средства этого языка не предназначены для описания процессов вычисления выражений, а только лишь фиксируют необходимость выполнения тех или иных условий применительно к отдельным

компонентам моделей ООАП. Главное назначение языка OCL — гарантировать справедливость ограничений для отдельных объектов модели.

Язык OCL может быть использован для решения следующих задач:

- описание инвариантов классификаторов и типов в модели классов;
- описание пред- и постусловий в операциях и методах;
- описание сторожевых условий элементов модели;
- спецификация ограничений на операции;
- в качестве языка запросов к элементам модели.

Язык OCL также применяется для спецификации базовой нотации в метамодели языка UML 2.0, в частности, для записи правил правильного построения выражений, определяющих семантику элементов основных пакетов языка UML 2.0.

Язык OCL является типизированным языком, поскольку каждое выражение языка OCL имеет некоторый тип. В правильно построенном выражении языка OCL все типы должны быть совместимы между собой. Например, нельзя в выражении сравнивать целое число со строкой. Типом может быть любой вид классификатора языка UML 2.0. При этом предполагается, что оценка выражений OCL выполняется мгновенным образом. Это означает, что состояние элементов модели не может изменяться в течение оценки того или иного выражения.

Описание языка OCL отличается от описаний традиционных формальных языков своим менее формальным характером. Базовым элементом языка OCL является выражение, которое строится по определенным правилам. При этом язык допускает расширение своих конструкций за счет включения в свой состав дополнительных типов.

П1.1. Выражения языка OCL

Выражение (expression) как элемент языка OCL является составной частью метамодели языка UML 2.0 и представляет собой утверждение относительно множества (возможно, пустого) экземпляров некоторого элемента модели. Выражение OCL является контекстно-зависимым в том смысле, что каждое выражение рассматривается только в том контексте или окружении, в котором определен соответствующий объект UML 2.0. С другой стороны, выражение не может модифицировать этот контекст или окружение объекта.

В модели выражение используется для записи некоторых условий, которым должны удовлетворять все экземпляры соответствующего классификатора.

В этом случае говорят, что выражение служит для представления некоторых инвариантных свойств соответствующих элементов модели.

Выражение состоит из тела и имени языка интерпретации, в рамках которого оценивается тело данного выражения. *Тело* выражения представляет собой строку текста без кавычек на некотором языке, возможно, некотором подмножестве естественного языка. Для записи выражения OCL рекомендуется использовать моноширинный шрифт. Например, строка текста, взятая без кавычек: `'Эта строка текста есть выражение языка OCL'` по определению является выражением OCL.

При записи выражений могут использоваться некоторые ключевые слова. Часть из них записывается полужирным шрифтом на языке оригинала. Например, ключевое слово **`context`** представляет контекст выражения. Ключевые слова **`inv`**, **`pre`** и **`post`** обозначают стереотипы, соответственно `<<инвариант>>`, `<<предусловие>>` и `<<постусловие>>` некоторого ограничения. Ключевые слова OCL для удобства чтения записываются полужирным шрифтом. Полужирный шрифт не имеет никакого формального смысла. OCL-выражения записываются только с использованием символов кодировки ASCII. Само выражение OCL записывается после двоеточия. Например:

```
context Имя_типа inv:
'Это выражение OCL со стереотипом "инвариант" в контексте Имя_типа'
```

Каждое выражение OCL должно быть записано в контексте некоторого типа или класса. Для обращения к контексту экземпляра типа используется ключевое слово `self`, записанное обычным шрифтом. Как правило, из записи выражения OCL непосредственно следует, о контексте какого экземпляра идет речь. Например, если контекст выражения — компания, то `self` относится к любому экземпляру компании.

Контекст выражения OCL внутри модели UML 2.0 может быть определен посредством явного *объявления контекста* в начале выражения OCL. Объявление контекста ограничения на диаграмме может быть представлено соответствующим стереотипом и пунктирной линией, связывающей это ограничение с контекстом.

Выражение OCL может быть частью записи некоторого инвариантного свойства или инварианта, записанное в форме ограничения со стереотипом `<<инвариант>>`. Выражение OCL, которое является инвариантом типа, должно быть истинно для всех экземпляров классификатора этого типа. Следует заметить, что все выражения OCL, которые выражают инварианты, имеют логический (или булев) тип, т. е. при своей оценке могут принимать только значения "истина" или "ложь".

Предположим, что в контексте некоторой компании (типа компании) следует определить инвариант, который устанавливает обязательное условие — любая отдельно взятая компания всегда должна содержать некоторое число служащих. Это можно записать в форме следующего выражения OCL:

```
self.numberOfEmployees > 0
```

Однако при записи инварианта в некотором контексте должно быть указано соответствующее ключевое слово **inv**, которое объявляет, что следующее за ним ограничение будет инвариантным, т. е. обязательным ограничением. Если инвариант ассоциируется с классификатором, то на него ссылаются как на некоторый "тип". Выражение OCL является инвариантом типа и должно быть истинным для всех экземпляров этого типа в любой момент времени.

Окончательная запись инвариантного ограничения для рассматриваемого примера будет иметь следующий вид:

```
context Компания inv:
        self.numberOfEmployees > 0
```

Если контекст выражения очевиден, то ключевое слово `self` может быть пропущено. Однако можно использовать и явное определение элемента, в контексте которого записывается выражение:

```
context c : Компания inv:
        c.numberOfEmployees > 0
```

Этот инвариант эквивалентен предыдущему. Дополнительно, но необязательно, после ключевого слова **inv** и перед двоеточием может быть написано имя ограничения, что позволяет ссылаться на это ограничение из других элементов модели. Так, если определить имя этого ограничения как `Персонал`, то оно может быть записано в следующем виде:

```
context c : Компания inv Персонал:
        c.numberOfEmployees > 0
```

П1.2. Основные типы значений и операций в языке OCL

В языке OCL ряд основных или базисных типов предопределен и всегда доступен разработчику модели. Эти предопределенные типы значений не зависят от конкретной объектной модели и не могут быть переопределены, поскольку являются частью определения языка OCL.

Основные типы значений языка OCL следующие:

- *булев* или *логический тип* — принимает значение "истина" или "ложь";

- *целое число* — может принимать любое значение из множества целых чисел, например: –5, 2, 34, 26 524;
- *действительное число* — может принимать любое значение из множества действительных чисел, например: 1,5; 3,14;
- *строка* — содержит любое количество символов текста, например: `'быть или не быть...'`.

В языке OCL также определены операции над этими основными типами, некоторые из них представлены далее:

- для булева типа определены операции: **`and`**, **`or`**, **`xor`**, **`not`**, **`implies`**, **`if-then-else-endif`**;
- для целых чисел определены арифметические операции `+`, `-`, `*`, `/`, **`div`** и операции **`abs`**, **`mod`**, **`max`**, **`min`**;
- для действительных чисел определены арифметические операции `+`, `-`, `*`, `/` и операции **`abs`**, **`floor`**, **`round`**, **`max`**, **`min`**;
- для типа *строка* определены операции: конкатенации **`concat`**, определения размера **`size()`**, преобразования регистра **`toUpper`** и **`toLower`**.

В выражениях языка OCL также используются операции сравнения и круглые скобки для указания последовательности выполнения этих операций. Последовательность выполнения операций определяется их приоритетом. По умолчанию в языке OCL установлен следующий приоритет операций (в порядке убывания приоритета):

- `@pre`;
- точка `.` или операция *стрелка* `->`;
- унарное логическое отрицание **`not`** и унарный минус `-`;
- умножение `*` и деление `/`;
- сложение `+` и бинарный минус `-`;
- логический условный оператор **`if-then-else-endif`**;
- операции сравнения: `<`, `<=`, `>`, `>=`;
- операции равенства `=` и неравенства `<>`;
- логические бинарные операции **`and`**, **`or`** и **`xor`**;
- логическая бинарная операция **`implies`**.

Для записи комментария применяется символ двойного минуса `--`. Например, следующая запись является комментарием языка OCL:

```
-- Это комментарий в языке OCL
```

П1.3. Операции над отдельными типами значений

В языке OCL в качестве основной используется *инфиксная* запись операций. Это означает, что второй аргумент бинарной операции рассматривается как ее параметр, а сама операция применяется к первому аргументу. При этом применение операции к этому аргументу обозначается точкой после этого аргумента. Таким образом, выражение `a+b` эквивалентно следующему выражению языка OCL: `a.+(b)`, где второй аргумент `b` выступает в роли параметра операции сложения. Тем не менее допускается использование и обычной записи, если это не вызывает недоразумений.

П1.3.1. Операции с действительными числами

Действительные числа являются наиболее общим типом чисел, которые используются в языке OCL. Формальная запись отдельной бинарной операции имеет следующий общий формат:

```
r1 'символ операции' r2 :'тип возвращаемого значения'
```

или в эквивалентной записи:

```
r1.'символ операции'(r2) :'тип возвращаемого значения'
```

Здесь `r1`, `r2` обозначают произвольные действительные числа, *`'символ операции'`* — условное обозначение операции, а *`'тип возвращаемого значения'`* должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для действительных чисел определены следующие операции сравнения.

- `r1 = r2 : Boolean`. Возвращает значение "истина", если значение `r1` в точности равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 = 3.14` принимает значение "истина", а выражение `3.14 = 3.1` принимает значение "ложь".
- `r1 <> r2 : Boolean`. Возвращает значение "истина", если значение `r1` не равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 <> 3.1` принимает значение "истина", а выражение `3.14 <> 3.14` принимает значение "ложь".
- `r1 < r2 : Boolean`. Возвращает значение "истина", если значение `r1` строго меньше значения `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.1 < 3.14` принимает значение "истина", а выражение `3.14 < 3.1` принимает значение "ложь".

- `r1 > r2 : Boolean`. Возвращает значение "истина", если значение `r1` строго больше значения `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 > 3.1` принимает значение "истина", а выражение `3.1 > 3.14` принимает значение "ложь".
- `r1 <= r2 : Boolean`. Возвращает значение "истина", если значение `r1` меньше или равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.1 <= 3.14` принимает значение "истина", а выражение `3.14 < = 3.1` принимает значение "ложь".
- `r1 >= r2 : Boolean`. Возвращает значение "истина", если значение `r1` больше или равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 >= 3.14` принимает значение "истина", а выражение `3.1 >= 3.14` принимает значение "ложь".

Далее для действительных чисел определены следующие арифметические операции:

- `r1 + r2 : Real`. Возвращает результат сложения двух действительных чисел. Например, выражение `3.14 + 3.14` принимает значение 6.28;
- `r1 - r2 : Real`. Возвращает результат вычитания двух действительных чисел. Например, выражение `3.14 - 3.14` принимает значение 0.0;
- `r1 * r2 : Real`. Возвращает результат умножения двух действительных чисел. Например, выражение `3.14 * 3.14` принимает значение 9.8596;
- `r1 / r2 : Real`. Возвращает результат деления двух действительных чисел. Например, выражение `3.14 / 3.14` принимает значение 1.0.

И, наконец, имеются некоторые дополнительные операции, которые могут применяться к действительным числам (здесь `r` — любое действительное число).

- `r.`**`abs`** `: Real`. Возвращает в качестве результата абсолютное значение действительного числа `r`. Другими словами, возвращает само значение `r`, если `r>=0`, и возвращает значение `-r`, если `r<0`. Например, выражение `3.14.`**`abs`** принимает значение 3.14, а выражение `(3.14-5.14).`**`abs`** принимает значение 2.
- `r.`**`floor`** `: Integer`. Возвращает в качестве результата наибольшее целое число, которое меньше или равно значению действительного числа `r`. Другими словами, возвращаемое значение меньше или равно `r` и одновременно строго больше `r-1`. Например, выражение `3.14.`**`floor`** принимает значение "3", а выражение `(3.14-5.1).`**`floor`** принимает значение –2.
- `r.`**`round`** `: Integer`. Возвращает в качестве результата целое число, которое является ближайшим к значению действительного числа `r`. Если такими

целыми являются сразу два числа, то возвращается наибольшее из них. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **post**: ((r-result).**abs**<0.5) **or** ((r-result).**abs**=0.5 **and** (result>r)). Здесь result служит для обозначения возвращаемого данной операцией значения. Например, выражение 3.14.round принимает значение "3", а выражение (3.14-5.64).**round** принимает значение –2.

- r1.**max**(r2) : Real. Возвращает в качестве результата наибольшее из двух действительных чисел r1 или r2. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **post**: **if** r1>=r2 **then** result=r1 **else** result=r2 **endif**. Здесь result также служит для обозначения возвращаемого данной операцией значения. Например, выражение (3.14).**max**(3.1) принимает значение 3.14.
- r1.**min**(r2) : Real. Возвращает в качестве результата наименьшее из двух действительных чисел r1 или r2. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **post**: **if** r1<=r2 **then** result=r1 **else** result=r2 **endif**. Здесь result также служит для обозначения возвращаемого данной операцией значения. Например, выражение (3.14).**min**(3.1) принимает значение 3.1.

Поскольку целые числа в языке OCL являются подтипом действительных чисел, то в рассмотренные выше операции в качестве действительных параметров могут входить отдельные целочисленные значения, что не противоречит общему определению этих операций.

П1.3.2. Операции с целыми числами

Для целых чисел вводятся в рассмотрение самостоятельные операции, в формальной записи которых используется общий формат бинарных операций:

```
i1 'символ операции' i2 :'тип возвращаемого значения'
```

или в эквивалентной записи:

```
i1.'символ операции'(i2) :'тип возвращаемого значения'
```

Здесь i1, i2 обозначают произвольные целые числа, *'символ операции'* — условное обозначение операции, а *'тип возвращаемого значения'* должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для действительных чисел определена операция равенства:

```
i1 = i2 : Boolean
```

Возвращает значение "истина", если значение `i1` в точности равно значению `i2`, в противном случае возвращает значение "ложь". Например, выражение `3 = 3` принимает значение "истина", а выражение `3 = 2` принимает значение "ложь".

Далее для целых чисел определены следующие арифметические операции:

- `i1 + i2 : Integer`. Возвращает результат сложения двух целых чисел. Например, выражение `3 + 2` принимает значение 5;
- `i1 - i2 : Integer`. Возвращает результат вычитания двух целых чисел. Например, выражение `3 - 4` принимает значение –1;
- `i1 * i2 : Integer`. Возвращает результат умножения двух целых чисел. Например, выражение `3 * 4` принимает значение 12;
- `i1 / i2 : Real`. Возвращает результат деления двух целых чисел в виде действительного числа. Например, выражение `3 / 2` принимает значение 1.5.

И, наконец, имеются некоторые дополнительные операции, которые могут применяться к целым числам (здесь `i` — любое целое число).

- `i.`**`abs`** `: Integer`. Возвращает в качестве результата абсолютное значение целого числа `i`. Другими словами, эта операция возвращает само значение `i`, если `i>=0`, и возвращает значение `-i`, если `i<0`. Например, выражение `3.`**`abs`** принимает значение 3, а выражение `(3-5).`**`abs`** принимает значение 2.
- `i1.`**`div`**`(i2) : Integer`. Возвращает в качестве результата целое число, равное частному от деления `i1` на `i2` с отброшенной дробной частью. Другими словами, перед выполнением этой операции должно быть справедливым предусловие: **`pre`**`: i2<>0` (делитель не должен быть равен 0), а после выполнения этой операции — постусловие: **`post`**`: `**`if`** `i1/i2>=0` **`then`** `result=(i1/i2).`**`floor`** **`else`** `result=-((-i1/i2).`**`floor`**`)` **`endif`**. Здесь `result`, как и ранее, служит для обозначения возвращаемого данной операцией значения. Например, выражение `3.`**`div`**`( 2)` принимает значение 1.
- `i1.`**`mod`**`(i2) : Integer`. Возвращает в качестве результата целое число, равное остатку от деления `i1` на `i2`. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **`post`**`: result=i1-(i1.div(i2))*i2`. Например, выражение `5.`**`mod`**`( 2)` принимает значение 1.
- `i1.`**`max`**`(i2) : Integer`. Возвращает в качестве результата наибольшее из двух целых чисел `i1` или `i2`. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **`post`**`: `**`if`** `i1>=i2`

then result=i1 **else** result=i2 **endif**. Например, выражение 3.**max**(1) принимает значение 3.

- i1.**min**(i2) : Integer. Возвращает в качестве результата наименьшее из двух целых чисел i1 или i2. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **post: if** i1<=i2 **then** result=i1 **else** result=i2 **endif**. Например, выражение 3.**min**(1) принимает значение 1.

П1.3.3. Операции со строками

Тип строки в языке OCL соответствует ASCII-строкам, т. е. в качестве отдельных символов строк необходимо использовать только символы кодировки ASCII. В этой связи резонно предположить, что имеется в виду расширенная кодировка ASCII, которая допускает применение символов кириллицы, хотя этот факт никак не отражается в исходной документации. Для формальной записи бинарных операций со строками используется общий формат:

```
s1 'символ операции' s2 :'тип возвращаемого значения'
```

или в эквивалентной записи:

```
s1.'символ операции'(s2) :'тип возвращаемого значения'
```

Здесь s1, s2 обозначают произвольные строки, *'символ операции'* — условное обозначение операции, а *'тип возвращаемого значения'* должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для строк определена операция равенства:

```
s1 = s2 : Boolean
```

Возвращает значение "истина", если строка s1 содержит в точности те же символы (включая их порядок и регистр), что и строка s2, в противном случае возвращает значение "ложь". Например, выражение 'Компания' = 'Компания' принимает значение "истина", а выражение 'Компания' ='компания' принимает значение "ложь".

Далее для строк определены следующие унарные операции:

- s.**size()** : Integer. Возвращает в качестве результата количество символов в строке s. При этом пробелы также должны учитываться. Например, выражение 'Компания'.**size()** принимает значение 8;
- s.**toUpper** : String. Возвращает в качестве результата строку, в которой все символы (буквы) нижнего регистра исходной строки s заменены аналогичными буквами верхнего регистра. При этом должно выполняться

следующее постусловие: **post**: result.**size()** = s.**size()**. Здесь result, как и ранее, служит для обозначения возвращаемого данной операцией значения. Например, выражение 'Компания'.**toUpper** принимает значение 'КОМПАНИЯ';

- s.**toLower** : String. Возвращает в качестве результата строку, в которой все символы (буквы) верхнего регистра исходной строки s заменены аналогичными буквами нижнего регистра. При этом должно выполняться следующее постусловие: **post**: result.**size()** = s.**size()**. Например, выражение 'Компания'.**toLower** принимает значение 'компания'.

И, наконец, для строк определены бинарная операция *конкатенации* и тернарная операция выделения подстроки заданной длины (здесь i1, i2 — произвольные целые числа).

- s1.**concat**(s2) : String. Возвращает в качестве результата строку, в которой вначале следуют все символы первой строки s1, а затем без пробела — все символы второй строки s2. При этом должны выполняться следующие постусловия:
  - **post**: result.**size()** = s1.size()+s2.**size()**;
  - **post**: result.**substring**(1, s1.**size()**)= s1;
  - **post**: result.**substring**(s1.**size()**+1, result.**size()**)= s2.

  Здесь используется операция выделения подстроки заданной длины substring, которая определяется ниже. Например, выражение 'Персональный'.**concat**(' Компьютер') принимает значение 'Персональный Компьютер'.
- s.**substring**(i1, i2) : String. Возвращает в качестве результата строку, которая является подстрокой исходной строки s, начиная символом с номером i1 и заканчивая символом с номером i2. Например, выражение 'Персональный'.**substring**(1, 7) принимает значение 'Персона'. Параметрами этой операции являются два целых числа, указывающие порядковый номер соответствующих символов.

П1.3.4. Операции с булевыми выражениями

Напомним, что булевы выражения могут принимать только одно их двух возможных значений: "истина" или "ложь". Для формальной записи бинарных операций с булевыми выражениями используется обычный неинфиксный формат, принятый в математической логике (b, b1, b2 — произвольные булевы выражения).

Прежде всего, для булевых выражений определена бинарная операция равенства:

```
b1=b2: Boolean
```

Возвращает в качестве результата значение "истина", если значения b1 и b1 одновременно принимают одно из значений — "истина" или "ложь", и значение "ложь" — в противном случае. Например, выражение (5<3) = (2=3) принимает значение "истина", а выражение (5<3) = (2<>3) принимает значение "ложь".

Далее, для булевых выражений определена унарная операция *логического отрицания*:

not b: Boolean

Возвращает в качестве результата значение "истина", если значение b — "ложь", и значение "ложь", если значение b — "истина". Другими словами, после выполнения этой операции должно выполняться следующее постусловие: **post**: **if** b="истина" **then** result="ложь" **else** result="истина" **endif**. Например, выражение **not** (5<3) принимает значение "истина".

Для булевых выражений определены следующие основные логические (бинарные) операции:

- b1 **or** b2: Boolean. Логическое "ИЛИ". Возвращает в качестве результата значение "истина", если значение "истина" имеет или b1, или b2, или они одновременно, и значение "ложь" — в противном случае. Например, выражение (5<3) **or** (2=3) принимает значение "ложь", а выражение (5<3) **or** (2<>3) принимает значение "истина";
- b1 **xor** b2: Boolean. Логическое "исключающее ИЛИ". Возвращает в качестве результата значение "истина", если значение "истина" имеет или b1, или b2 (но не одновременно), и значение "ложь" — в противном случае. Для этой операции должно выполняться следующее постусловие: **post**: (b1 or b2) **and not**(b1 = b2). Здесь используется операция логического "И" (**and**), которая определяется далее. Например, выражение (5<3) **xor** (2=3) принимает значение "ложь", а выражение (5<3) **xor** (2<3) принимает значение "истина";
- b1 **and** b2: Boolean. Логическое "И". Возвращает в качестве результата значение "истина", если значение "истина" имеют b1 и b1 одновременно, и значение "ложь" — в противном случае. Например, выражение (5<3) **and** (2=3) принимает значение "ложь", а выражение (5>3) **and** (2<>3) принимает значение "истина".

И, наконец, для логических выражений определены бинарная операция логической *импликации* и упоминавшаяся ранее специальная логическая конструкция **if-then-else-endif**.

- b1 **implies** b2: Boolean. Логическая импликация. Возвращает в качестве результата значение "истина", если b1 имеет значение "ложь", или если

`b1` и `b2` имеют значение "истина" одновременно, и значение "ложь" — в противном случае. Например, выражение `(5<3)` **`implies`** `(2=3)` принимает значение "истина", а выражение `(5>3)` **`implies`** `(2=3)` принимает значение "ложь".

- **`if`** `b` **`then`** `e1` **`else`** `e2` **`endif`** : `тип e1 или e2`. Логическая условная конструкция. Возвращает в качестве результата выражение `e1`, если логическое выражение `b` имеет значение "истина", и возвращает в качестве результата выражение `e2`, если логическое выражение `b` имеет значение "ложь". Например, рассмотренное ранее постусловие для логического отрицания **`post`**: **`if`** `b="истина"` **`then`** `result="ложь"` **`else`** `result="истина"` **`endif`**. Здесь `e1` и `e2` — произвольные выражения языка OCL.

П1.3.5. Операция *@pre* для указания предшествующих элементов

Особенность записи постусловий для операций заключается в том, что они могут ссылаться на значения элементов языка OCL, которые имели место до завершения соответствующей операции или метода. В этом случае возникает необходимость явно указать то значение элемента, которое он имел до начала выполнения операции. Для этой цели в языке OCL используется специальное постфиксное ключевое слово `@pre`. Постфиксный характер этого слова состоит в том, что оно всегда записывается после имени элемента без пробела.

Например, если необходимо записать постусловие для операции определения возраста сотрудника компании в текущем году, то это можно сделать следующим образом:

```
context Сотрудник::возраст
post: if t_Date >= b_Date then возраст = возраст@pre+1
      else возраст = возраст@pre endif.
```

Здесь `t_Date` обозначает текущую дату (день, месяц), `b_Date` — дату рождения сотрудника (день, месяц). Предполагается, что обе эти даты преобразованы к типу целое число.

П1.4. Допустимые выражения в языке OCL

Поскольку OCL — язык с контролем типа выражений, все его выражения должны быть согласованными с основными или производными типами значений. Например, нельзя сравнивать целое число с булевым типом или строкой. Выражение OCL, в котором все типы согласованы между собой, называется *допустимым* выражением языка OCL. Выражение OCL, в котором

некоторые типы не согласованы — недопустимое выражение. Последний случай является ошибкой согласованности типов.

Существуют два правила согласованности типов выражений:

- каждый тип соответствует каждому из своих супертипов;
- согласованность типов транзитивна: если тип 1 соответствует типу 2, и тип 2 соответствует типу 3, то тип 1 соответствует типу 3.

Примеры допустимых выражений языка OCL:

```
1 + 2 * 34
12 + 13.5
13 < 2
```

Примеры недопустимых выражений языка OCL:

```
1 + 'мотоцикл'
23 * ложь
```

П1.5. Неопределенное выражение

Всякий раз, когда оценивается выражение OCL, существует возможность, что значение одного или нескольких элементов в выражении могут быть не определены. Если дело обстоит так, то полное выражение будет иметь значение "неопределенное".

Однако из этого правила имеются два исключения для следующих логических операций:

- выражение в целом принимает значение "истина", если хотя бы один логический операнд в операции **or** принимает значение "истина";
- выражение в целом принимает значение "ложь", если хотя бы один логический операнд в операции **and** принимает значение "ложь".

Вышеупомянутые два правила применимы независимо от порядка записи операндов, при этом значения отдельных операндов в логическом выражении могут быть неизвестны.

П1.6. Коллекции значений в языке OCL

Язык OCL содержит развитые средства для представления различных коллекций допустимых значений типов. При этом может учитываться порядок следования значений или повторение отдельных значений в коллекции. В языке OCL различаются типы коллекций значений.

- *Множество* (set) — представляет собой неупорядоченный список отдельных значений основного типа без повторяющихся элементов. Пример

множества, состоящего из положительных целых чисел: `{1, 6, 4, 8, 2, 5, 10}`.

- *Упорядоченное множество* (ordered set) — представляет собой упорядоченный список отдельных значений основного типа без повторяющихся элементов. Пример множества, состоящего из положительных целых чисел: `{1, 2, 4, 5, 6, 8, 10}`.
- *Мультимножество* (bag) — представляет собой коллекцию отдельных значений основного типа, элементы в которой могут повторяться. Пример мультимножества, состоящего из положительных целых чисел: `{1, 2, 1, 4, 5, 8, 6, 8, 1, 10}`. Мультимножество, которое в математике называют также комплектом, представляет собой неупорядоченный список элементов.
- *Последовательность* (sequence) — представляет собой упорядоченный список отдельных значений основного типа с возможно повторяющимися элементами. Пример последовательности, состоящей из положительных целых чисел: `{1, 2, 4, 5, 6, 8, 10}`. Коллекция чисел `{0, 1, 3, 3, 7, 8, 8}` также является последовательностью.

Для определения совместимости этих типов необходимо придерживаться следующего правила: типы "множество", "упорядоченное множество", "мультимножество" и "последовательность" являются подтипами типа "коллекция".

П1.7. Операции над коллекциями значений

Над коллекциями значений могут быть выполнены некоторые операции, такие как операция *выбора* (**select**) или *исключения* (**reject**), а также операции **collect**, **forAll**, **exists**.

П1.7.1. Операция выбора *select*

Операция выбора используется для спецификации части коллекции, элементы которой удовлетворяют некоторому логическому условию. Общий формат записи этой операции имеет следующий вид:

Коллекция->**select** (*Логическое выражение*)

В результате применения этой операции возвращаются все элементы исходной коллекции, для которых *Логическое выражение* имеет значение "истина". Например, следующее выражение языка OCL специфицирует обязательное

требование, которое утверждает, что среди сотрудников компании не должно быть сотрудников, возраст которых не превышает 18 лет:

```
context Компания inv:
        self.сотрудник -> select(возраст < 18) -> isEmpty
```

В этом случае операция выбора проверяет условие выбора последовательно для всех сотрудников компании. Если это условие выполняется, то сотрудник попадает в формируемую операцией выбора коллекции, в противном случае — нет. При этом ключевое слово **isEmpty** (пусто) означает, что возвращаемая операцией коллекция не должна содержать ни одного элемента. В противном случае она считается пустой коллекцией, по аналогии с пустым множеством. С этой операцией также может быть использоваться ключевое слово **notEmpty** (не пусто), которое означает, что возвращаемая операцией коллекция должна содержать хотя бы один элемент.

П1.7.2. Операция исключения *reject*

Операция исключения возвращает коллекцию значений, для которых логическое выражение принимает значение "ложь". Формат записи этой операции такой же, как и для операции выбора. Например, с помощью этой операции можно специфицировать условие, согласно которому все сотрудники компании должны быть семейными людьми:

```
context Компания inv:
          self.сотрудник -> reject(имеет семью) -> isEmpty
```

П1.7.3. Операция формирования коллекции *collect*

Рассмотренные ранее операции выбора и исключения всегда возвращают в качестве результата часть исходной коллекции значений (возможно, пустую). Если необходимо специфицировать коллекцию значений, которая получается из некоторой другой коллекции на основе проверки условия для отдельного свойства, то используется операция **collect**. Эта операция позволяет сформировать коллекцию значений, которые отличаются от значений исходной коллекции, но контекстно связаны с ней.

Формат применения этой операции следующий:

Исходная_коллекция->**collect**(*выражение*)

Например, если необходимо сформировать коллекцию дат принятия на работу сотрудников компании, то это можно сделать с помощью следующего выражения языка OCL:

```
self.сотрудник->collect(Дата_принятия_на_работу)
```

В результате применения этой операции будет сформирована коллекция, значениями которой будут являться даты приема на работу сотрудников рассматриваемой компании. Следует заметить, что тип значений возвращаемой этой операцией коллекции может отличаться от типа значений исходной коллекции.

П1.7.4. Операция "для всех" *forAll*

Некоторые типы ограничений могут относиться сразу ко всем элементам заданной коллекции. Для этого случая в языке OCL существует специальная операция **forAll**. Эта операция позволяет специфицировать некоторое логическое выражение, которое должно быть справедливым для всех элементов коллекции. Общий формат записи операции **forAll** имеет следующий вид:

Коллекция->**forAll**(*Логическое выражение*): Boolean

Эта операция возвращает значение "истина", если *Логическое выражение* справедливо для всех без исключения элементов исходной коллекции, и возвращает значение "ложь" в противном случае. Операция **forAll** может быть использована для записи инвариантов. Например, если в контексте некоторого банка счет клиента не может быть меньше $100, то это условие (инвариант) можно записать в следующем виде:

```
context Б : Банк inv:
          self.клиенты->forAll(счет >= 100$).
```

Очевидно, что если хотя бы для одного клиента это условие не будет выполнено, то данное инвариантное условие будет нарушено, что недопустимо для рассматриваемой модели. В некотором смысле операция **forAll** выполняет роль квантора всеобщности математической логики, однако достоинством языка OCL является возможность ее записи в форме операции.

Другим примером может служить запись распространенного инварианта, который требует единственности имен, в частности — имен клиентов банка:

```
context Б : Банк inv:
          self.клиенты->forAll(к1, к2 : клиент | к1<>к2
          implies к1.имя<>к2.имя).
```

П1.7.5. Операция "существует" *exists*

Как следует из названия этой операции, она применяется для спецификации ситуации, при которой исходная коллекция содержит хотя бы один элемент, удовлетворяющий некоторому логическому выражению. Общий формат записи операции **exists** имеет следующий вид:

Коллекция->**exists** (*Логическое выражение*): Boolean

Эта операция возвращает значение "истина", если *Логическое выражение* справедливо хотя бы для одного (а может быть и нескольких) элемента исходной коллекции, и возвращает значение "ложь" в противном случае. Операция **exists** также может быть использована для записи инвариантов.

Примером использования этой операции может служить ситуация, при которой определяется контрольный пакет акций некоторой компании или акционерного общества. Если контрольный пакет характеризуется наличием у отдельного акционера 50% и более от общего числа акций, то это условие (инвариант) может быть записано в следующем виде:

```
context к : Компания inv Контрольный пакет акций:
          self.акционеры->exists(а : акционер | а.количество_акций >=
          50% от общего числа акций)
```

Или более строго с использованием операций с целыми числами:

```
context к : Компания inv Контрольный пакет акций:
          self.акционеры->exists(а : акционер | а.количество_акций >=
          к.количество_акций.div(2))
```

В этом примере инвариант имеет собственное имя, на которое можно ссылаться из других компонентов модели. Следует отметить, что операция **exists** также выполняет роль квантора математической логики, но возможность ее записи в форме операции обладает целым рядом удобств.

П1.7.6. Другие операции над коллекциями значений

Поскольку коллекция в языке OCL обобщает типы множества, упорядоченного множества, мультимножества и последовательности, для нее, как для абстрактного супертипа, определены следующие операции.

- `коллекция->count(объект_OCL) : Integer`. Эта операция подсчитывает количество вхождений заданного элемента в исходную коллекцию. Операция **count** возвращает целое число, равное количеству вхождений `объекта_OCL` в эту коллекцию. Например, если исходная коллекция равна `коллекция ={2, 4, 5, 4, 6, 7, 4, 9, 2}`, то выражение `коллекция->count(4)` равно значению 3. С другой стороны, выражение `коллекция->count(3)` равно значению 0, поскольку число 3 не входит в эту коллекцию.
- `коллекция->includes(объект_OCL) : Boolean`. Эта операция проверяет наличие в исходной коллекции конкретного объекта языка OCL. Операция **includes** возвращает значение "истина", если `объект_OCL` является

элементом исходной коллекции, и значение "ложь" — в противном случае. Для этой операции должно выполняться следующее постусловие: **post**: result = (коллекция->**count**(объект_OCL)>0).

- коллекция->**excludes**(объект_OCL) : Boolean. Эта операция проверяет отсутствие в исходной коллекции конкретного объекта языка OCL. Операция **excludes** возвращает значение "истина", если объект_OCL не является элементом исходной коллекции, и значение "ложь" — в противном случае. Для этой операции должно выполняться следующее постусловие: post: result = (коллекция->**count**(объект_OCL)=0).

П1.8. Некоторые операции с множествами, последовательностями и комплектами

Для множеств в языке OCL определены основные теоретико-множественные операции: объединение, пересечение, разность множеств. При определении этих операций set1, set2, set3 обозначают произвольные множества языка OCL.

- set1->**union**(set2) : set3. Операция *объединения* множеств. В качестве результата возвращает множество set3, которое представляет собой теоретико-множественное объединение исходных множеств set1 и set2. Множество set3 состоит из элементов, которые принадлежат или множеству set1, или множеству set2, или им обоим.
- set1->**intersection**(set2) : set3. Операция *пересечения* множеств. В качестве результата возвращает множество set3, которое представляет собой теоретико-множественное пересечение исходных множеств set1 и set2. Множество set3 состоит из элементов, которые одновременно принадлежат множеству set1 и множеству set2.
- set1 - set2 : set3. Операция *разности* множеств. В качестве результата возвращает множество set3, которое представляет собой теоретико-множественную разность исходных множеств set1 и set2. Множество set3 состоит из элементов, которые принадлежат множеству set1 и не принадлежат множеству set2.

Аналогичные операции определяются применительно к последовательностям и комплектам, однако их семантика является более сложной. В частности, операции пересечения и объединения последовательностей имеют несколько постусловий, каждое из которых отражает отдельный аспект получаемых результатов.

П1.9. Операции преобразования типов

Поскольку множество, последовательность и мультимножество являются подтипами типа "коллекция", некоторые из них можно преобразовывать друг в друга. Речь идет о том, что каждая конкретная коллекция может быть записана в форме множества, хотя при этом возможна некоторая потеря информации. В форме множества может быть записана коллекция и мультимножество. Далее приводятся операции, которые позволяют преобразовывать различные подтипы коллекции.

- `мультимножество->asSet : set`. Операция **asSet** позволяет преобразовать мультимножество в множество. Эта операция возвращает в качестве результата множество `set`, состоящее из элементов комплекта без повторений. Например, если мультимножество равно `{1, 3, 4, 3, 5, 2, 1}`, то выражение `мультимножество->asSet` равно значению `{1, 3, 4, 5, 2}` или значению `{1, 2, 3, 4, 5}`. Следует отметить, что порядок записи элементов этого множества не имеет значения, поэтому оба полученных результата являются эквивалентными.
- `последовательность->asSet : set`. Операция **asSet** также позволяет преобразовать последовательность в множество. Эта операция возвращает в качестве результата множество `set`, состоящее из элементов исходной последовательности без учета порядка и без повторений. Например, если последовательность равна `{1, 1, 2, 3, 3, 4, 5}`, то выражение `последовательность->asSet` равно значению `{1, 3, 4, 5, 2}` или значению `{1, 2, 3, 4, 5}`. При этом оба полученных результата являются эквивалентными.
- `последовательность->asBag : bag`. Операция **asBag** позволяет преобразовать последовательность в мультимножество. Эта операция возвращает в качестве результата мультимножество `bag`, состоящее из всех элементов исходной последовательности без учета порядка, но с учетом повторений. Например, если последовательность равна `{1, 1, 2, 3, 3, 4, 5}`, то выражение `последовательность->asBag` равно значению `{1, 3, 1, 4, 3, 5, 2}` или значению `{1, 3, 2, 3, 1, 4, 5}`. При этом оба полученных результата также являются эквивалентными.

Аналогично определяются и другие операции преобразования типов: `мультимножество->asSequence`, `множество->asSequence`, `множество->asBag`.

П1.10. Примеры записи выражений языка OCL

П1.10.1. Определение значения переменной

Для определения отдельной переменной используется оператор **let**. В следующем примере определяется переменная доход_за_год для класса Сотрудник:

```
context Сотрудник inv:
        let доход_за_год: Integer = self.работа.зарплата*12
```

В этом случае доход сотрудника за год полностью определяется его зарплатой.

П1.10.2. Определение возраста сотрудника

Для определения возраста сотрудника необходимо выполнить простое арифметическое действие — вычесть год рождения сотрудника из текущего года:

```
context Сотрудник inv:
        let возраст : Integer = текущая_дата.год - self.год_рождения
```

П1.10.3. Определение кратности значений

Для определения кратности значений используется операция **size()**. Например, чтобы задать ограничение для класса компании, у которой может быть единственный директор, можно использовать следующее выражение языка OCL:

```
context Компания inv:
        self.директор ->size() = 1
```

П1.10.4. Определение коллекции инвариантов

Наиболее распространенным способом применения языка OCL является спецификация коллекции инвариантов для одного и того же класса. В этом случае контекст выражения записывается только один раз, после чего следуют записи всех относящихся к этому контексту выражений. Например, следующие выражения используются для определения коллекции инвариантов для класса Компания в конкретной модели:

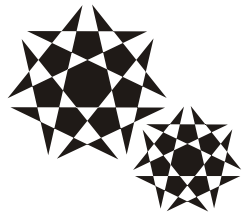
```
context Компания
        inv: self.сотрудник ->size() > 0
        inv: self.директор ->size() = 1
        inv: self.директор.возраст >= 30
        inv: self.сотрудник >= forAll(зарплата >= 500$)
```

Последнее условие определяет ограничение, согласно которому зарплата любого сотрудника такой компании не может быть менее $500.

В заключение рассмотрения языка OCL следует привести пример записи ограничений в метамодели самого языка UML 2.0. Так, например, условия уникальности имен экземпляров класса и характеристик классов могут быть записаны в форме следующих инвариантов:

```
context Class inv:
        Class.allInstances()->forAll(p1, p2 | p1.name = p2.name
        implies p1 = p2)
context Class inv:
        Class.feature->forAll(f1, f2 | f1.name = f2.name
        implies f1 = f2)
```

Здесь используется специальная предварительно определенная характеристика классов, интерфейсов и перечислений **allInstances()**, которая имеет своим результатом множество всех экземпляров соответствующего типа. Аналогично могут быть записаны и другие ограничения, которым должны удовлетворять различные элементы метамодели языка UML 2.0.



Приложение 2

Глоссарий

Настоящий глоссарий поясняет содержательный смысл всех понятий языка UML 2.0, которые используются для спецификации различных представлений модели и построения канонических диаграмм. Все термины расположены в алфавитном порядке, в скобках приводится эквивалент каждого из терминов на языке оригинала.

***n*-арная ассоциация** (n-ary association) — ассоциация между тремя и большим числом классов. Каждый экземпляр такой ассоциации представляет собой упорядоченный набор (кортеж), содержащий *n* экземпляров соответствующих классов.

Абстрактный класс (abstract class) — класс, который не имеет экземпляров или объектов времени выполнения.

Абстракция (abstraction) — характеристика сущности, которая отличает ее от других сущностей. Абстракция определяет границу представления соответствующего элемента модели.

Агрегат (aggregate) — класс, который представляет "целое" в отношении агрегации (в отношении типа "часть — целое").

Агрегация (aggregation) — направленное отношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности.

Актер (actor) — любая внешняя по отношению к проектируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач.

Активация (activation) — специальный символ на диаграмме последовательности, указывающий период времени, в течение которого линия жизни

выполняет некоторое действие. В языке UML 1.*x* этим термином назывался фокус управления (focus of control).

Активный класс (active class) — класс, каждый экземпляр которого имеет собственную нить управления.

Активный объект (active object) — объект, который имеет собственную нить управления и может инициировать деятельность по управлению. Является экземпляром активного класса.

Анализ (analysis) — отдельный этап процесса разработки программной системы, главная цель которого — построить модель предметной области решаемой проблемы, которая является независимой от особенностей реализации. Анализ раскрывает смысл того, что необходимо делать, а проектирование — как это следует делать. Противоположность: проектирование.

Артефакт (artifact) — элемент модели, который специфицирует некоторую физически существующую часть информации, используемую или производимую в ходе разработки программного обеспечения или в процессе развертывания и функционирования системы.

Архитектура (architecture) — организационная структура и связанное с ней поведение системы. Архитектура допускает рекурсивную декомпозицию на части, взаимодействующие посредством своих интерфейсов, на отношения, связывающие эти части, и на ограничения, соединяющие эти части в систему.

Аспект поведения модели (behavioral model aspect) — аспект модели, который рассматривает поведение отдельных элементов системы, включая методы, кооперации и состояния

Ассоциация (association) — произвольное отношение между двумя и более классификаторами, которое специфицирует факт наличия некоторой связи между соответствующими экземплярами этих классификаторов.

Ассоциация класс (association class) — элемент модели, который имеет свойства, как ассоциации, так и класса, и предназначенный для спецификации дополнительных свойств ассоциации в форме атрибутов и, возможно, операций класса.

Атрибут (attribute) — структурная характеристика классификатора, описывающая множество значений, которые могут принимать отдельные экземпляры классификатора.

Бизнес-актер (business actor) — индивидуум, группа, организация, компания или система, которые являются инициаторами бизнес-вариантов использования для моделируемой системы, но организационно в нее не входят.

Бизнес-вариант использования (business use case) — вариант использования, определяющий последовательность действий моделируемой системы, направленных на выполнение отдельного бизнес-процесса.

Бизнес-сущность (business entity) — пассивный класс, который не инициирует никаких сообщений на диаграммах взаимодействия и информация об экземплярах которого должна сохраняться в моделируемой системе в качестве результата выполнения бизнес-процесса.

Бинарная ассоциация (binary association) — ассоциация между двумя классами. Является специальным случаем *n*-арной ассоциации.

Булев тип (boolean type) — перечисление, имеющее только два значения: истина и ложь.

Булево выражение (boolean expression) — выражение, которое может принимать только одно из булевых значений.

Вариант использования (use case) — общая спецификация совокупности выполняемых системой действий с целью предоставления некоторого наблюдаемого результата, который имеет значение для одного или нескольких актеров.

Вершина (vertex) — в контексте языка UML 2.0 обозначает источник или цель перехода в конечном автомате. Вершиной может быть состояние или псевдосостояние.

Вершина разделения (fork vertex) — псевдосостояние, предназначенное для разделения входящего перехода на два или более перехода, которые имеют в качестве своих целей вершины в ортогональных регионах композитного состояния.

Вершина слияния (join vertex) — псевдосостояние, предназначенное для соединения нескольких переходов, которые имеют в качестве своих источников вершины из различных ортогональных регионов композитного состояния.

Взаимодействие (interaction) — единица поведения некоторого классификатора, которая концентрирует внимание на наблюдаемом обмене информацией между элементами, являющимися участниками этого взаимодействия.

Видимость (visibility) — совокупность конкретных значений (`public`, `protected` или `private`), характеризующих способность оказывать влияние на элемент модели со стороны других объектов модели.

Включение (include) — отношение зависимости, которое специфицирует тот факт, что некоторый вариант использования содержит поведение, определенное в другом варианте использования.

Внутренний переход (internal transition) — переход, специфицирующий реакцию на некоторое событие без изменения состояния.

Временная диаграмма (timing diagram) — диаграмма взаимодействия, которая служит для представления изменения состояния отдельных линий жизни или особенностей синхронизации сообщений во времени.

Временное выражение (time expression) — спецификация некоторого значения времени.

Временное ограничение (time constraint) — специальное ограничение, записанное в форме временного интервала.

Временное событие (time event) — элемент модели, который специфицирует некоторый момент времени посредством временного выражения.

Временной интервал (time interval) — диапазон между двумя временными выражениями.

Входное действие (entry action) — действие, которое выполняется в момент перехода конечного автомата в новое состояние.

Входной контакт (input pin) — узел объекта, который принимает значения от других действий в форме потока объектов.

Выражение (expression) — строка текста, при оценивании которой должно получиться значение определенного типа. Например, выражение `(7 + 5 * 3)` оценивается как целое число.

Выражение времени (time expression) — выражение, которое принимает значение абсолютного или относительного времени.

Выражение типа (type expression) — выражение, результат вычисления или оценки которого является ссылкой на один или несколько типов.

Выходной контакт (output pin) — узел объекта, который поставляет значения другим действиям в форме потока объектов.

Граничный класс (boundary class) — класс, который располагается на границе системы с внешней средой и непосредственно взаимодействует с актерами.

Действие (action) — представляет собой элементарную единицу спецификации поведения, которая не может быть далее декомпозирована в форме деятельности.

Действие на входе (entry action) — действие, выполняемое при входе в состояние конечного автомата.

Действие на выходе (exit action) — действие, выполняемое при выходе из состояния конечного автомата.

Действие наблюдения времени (time observation action) — действие, которое наблюдает текущую точку во времени и записывает это значение в некоторую структурную характеристику.

Действие наблюдения продолжительности (duration observation action) — действие, которое наблюдает продолжительность во времени и записывает это значение в некоторую структурную характеристику.

Действие передачи сигнала (send signal action) — действие, которое на основе своих входов создает экземпляр сигнала и передает его объекту цели.

Действие приема события (accept event action) — действие, которое ожидает наступление некоторого события.

Декомпозиция части (part decomposition) — элемент модели, предназначенный для представления внутренних взаимодействий одной из линий жизни, класс которой имеет собственную композитную структуру.

Делегирующий соединитель (delegation connector) — соединитель, который связывает внешний контракт компонента с реализацией этого поведения внутренними частями этого компонента.

Деятельность (activity) — спецификация параметризованного поведения в форме координируемой последовательности подчиненных единиц, индивидуальными элементами которых являются действия.

Диаграмма (diagram) — графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика. Нотация канонических диаграмм является основным средством разработки моделей на языке UML 2.0.

Диаграмма вариантов использования (use case diagram) — диаграмма концептуального уровня, которая служит для представления актеров и вариантов использования системы, а также отношений между ними. В нотации UML 2.0 варианты использования изображаются внутри границ субъекта, а актеры изображаются за пределами этой границы.

Диаграмма взаимодействия (interaction diagram) — общее название пяти канонических диаграмм, предназначенных для представления различных аспектов взаимодействия.

Диаграмма деятельности (activity diagram) — диаграмма, которая изображает поведение объекта или системы с использованием моделей потока данных и потока управления.

Диаграмма классов (class diagram) — диаграмма, которая служит для представления совокупности декларативных или статических элементов модели, таких как классы, типы, а также разнообразных отношений между ними.

Диаграмма коммуникации (communication diagram) — диаграмма, которая служит для представления взаимодействия между линиями жизни объектов в форме элементов внутренней структуры системы и передаваемых между ними сообщений. По форме совпадает с диаграммой кооперации языка UML 1.*x*.

Диаграмма композитной структуры (composite structure diagram) — диаграмма, которая изображает внутреннюю структуру классификаторов, таких как класс, компонент или кооперация, включая точки взаимодействия классификатора с другими частями системы.

Диаграмма компонентов (component diagram) — диаграмма физического уровня, которая служит для представления программных компонентов и зависимостей между ними.

Диаграмма конечного автомата (state machine diagram) — диаграмма, которая служит для представления дискретного поведения, моделируемого посредством переходов в системах с конечным числом состояний. По форме совпадает с диаграммой состояний (statechart diagram) языка UML 1.*x*, но содержит дополнительные элементы.

Диаграмма обзора взаимодействия (interaction overview diagram) — вариант диаграммы деятельности, которая служит для представления только взаимодействия потока управления в некоторой агрегированной форме.

Диаграмма объектов (object diagram) — диаграмма, которая служит для представления объектов и отношений меду ними в конкретный момент времени. Она может рассматриваться как специальный случай диаграммы классов или диаграммы коммуникации.

Диаграмма пакетов (package diagram) — диаграмма логического уровня, которая служит для представления организации элементов модели по пакетам, а также зависимостей между пакетами, включая импорт пакетов и слияние пакетов.

Диаграмма последовательности (sequence diagram) — диаграмма, которая служит для представления взаимодействия элементов модели в форме последовательности сообщений и соответствующих событий на линиях жизни объектов.

Диаграмма развертывания (deployment diagram) — диаграмма физического уровня, которая предназначена для представления общей конфигурации или топологии распределенной программной системы и содержит изображение размещения различных артефактов по отдельным узлам системы.

Дуга деятельности (activity edge) — абстрактный класс для направленных соединений между двумя узлами деятельности.

Зависимость (dependency) — форма взаимосвязи между двумя элементами модели, предназначенная для спецификации того обстоятельства, что изменение одного элемента модели приводит к изменению некоторого другого элемента.

Значение (value) — отдельный элемент из допустимой области значений переменной заданного типа.

Импорт пакета (package import) — направленное отношение между пакетами, при котором члены одного пакета могут быть добавлены в пространство имен другого пакета.

Импорт элемента (element import) — направленное отношение между импортирующим пространством имен и отдельным элементом пакета, которое позволяет ссылаться на этот элемент с использованием неквалифицированного имени.

Имя (name) — строка текста, которая используется для идентификации элемента модели.

Инвариант состояния (state invariant) — ограничение времени выполнения, которое должно быть выполнено для отдельных участников взаимодействия.

Инкапсуляция (encapsulation) — характеризует сокрытие отдельных деталей внутреннего устройства классов или компонентов от внешних по отношению к нему объектов или пользователей.

Инстанцирование (binding) — создание элемента модели на основе шаблона посредством подстановки аргументов вместо параметров шаблона.

Интервал (interval) — диапазон между двумя спецификациями значений.

Интервал продолжительности (duration interval) — диапазон между двумя продолжительностями.

Интерфейс (interface) — вид класса, который представляет собой объявление множества общедоступных характеристик и обязанностей.

Использование взаимодействия (interaction use) — элемент модели, представляющий параметризованную ссылку на некоторое взаимодействие в контексте другого взаимодействия.

Класс (class) — описание множества однородных объектов, имеющих одинаковые атрибуты, операции, методы, отношения с другими объектами и семантику.

Класс-сущность (entity class) — пассивный класс, информация о котором должна храниться постоянно и не уничтожаться с выключением системы.

Классификатор (classifier) — специальное понятие, предназначенное для классификации экземпляров, которые имеют общие характеристики. От классифика-

торов в метамодели языка UML 2.0 наследуют классы, интерфейсы, типы данных, кооперации, компоненты и некоторые другие элементы.

Классификация (classification) — установление соответствия между некоторым объектом и классификатором.

Клиент (client) — классификатор, который требует некоторого сервиса от другого классификатора.

Комбинированный фрагмент (combined fragment) — элемент модели, предназначенный для представления внутренней логической структуры фрагментов взаимодействия.

Композитное состояние (composite state) — состояние, содержащее в своем составе один регион или несколько ортогональных регионов.

Композиция (composition) — композитная агрегация, предназначенная для спецификации более сильной формы отношения "часть — целое", когда с уничтожением объекта класса-контейнера уничтожаются и все объекты, являющиеся его составными частями.

Компонент (component) — элемент модели, представляющий некоторую модульную часть системы с инкапсулированным содержимым, спецификация которого является взаимозаменяемой в его окружении.

Конец ассоциации (association end) — концевая точка ассоциации, которая связывает ассоциацию с классификатором.

Конец связи (link end) — отдельный экземпляр конца ассоциации или конец связи.

Конечный автомат (state machine) — модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом некоторых действий, а также изменение его отдельных свойств.

Конкретный класс (concrete class) — класс, на основе которого могут быть непосредственно созданы экземпляры (объекты).

Контейнер (container) — экземпляр или компонент, который включает в качестве своих частей другие экземпляры или компоненты.

Контекст (context) — представление множества относящихся к модели элементов, выполненное с определенной целью.

Кооперация (collaboration) — классификатор, предназначенный для описания некоторой структуры элементов или ролей, которые выполняют специализированные функции и совместно производят желаемую функциональность.

Кратность (multiplicity) — спецификация области значений допустимой мощности, которой могут обладать соответствующие множества.

Линия жизни (lifeline) — индивидуальный участник или сущность взаимодействия. В частном случае представляет собой экземпляр класса.

Манифестация (manifestation) — отношение для спецификации конкретного физического воплощения одного или нескольких элементов модели посредством артефакта.

Маркер (token) — элемент модели, предназначенный для представления некоторого объекта, данных или управления и существующий на диаграмме деятельности в отдельном узле.

Метакласс (metaclass) — класс, экземпляры которого также являются классами. Метаклассы используются обычно для построения метамоделей.

Мета-метамодель (meta-metamodel) — абстрактная модель, предназначенная для определения языка описания метамодели. Отношение между мета-метамоделью и метамоделью аналогично отношению между метамоделью и моделью.

Метамодель (metamodel) — модель, предназначенная для определения языка описания модели.

Метод (method) — конкретная реализация операции. Как правило, метод специфицирует конкретный алгоритм или процедуру, ассоциированную с соответствующей операцией.

Методология (methodology) — совокупность принципов и методик разработки программных систем и приложений, определяющих процесс построения моделей при выполнении соответствующих проектов.

Модель (model) — абстракция произвольной системы или объекта, рассматриваемая с определенной точки зрения и представленная на некотором языке или в графической форме. Одна и та же физическая система может быть представлена несколькими моделями, что отражает общий принцип многомодельности системного анализа.

Модель вариантов использования (use case model) — модель, которая описывает функциональные требования к системе в терминах вариантов использования.

Модель поведения (behavioral model) — модель, предназначенная для описания процесса функционирования системы, взаимодействия отдельных ее элементов во времени, а также процесса изменения состояний объектов и системы в целом.

Модуль (module) — часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

Мощность множества (cardinality) — число элементов конечного множества. Для бесконечных множеств имеет более сложную семантику.

Наследование (inheritance) — принцип, в соответствии с которым знание о более общей категории разрешается применять для более частной категории.

Наследование интерфейса (interface inheritance) — наследование интерфейса более общего элемента. Не включает в себя наследование реализации этого элемента.

Начальный узел (initial node) — узел управления, в котором начинается поток при вызове деятельности.

Начальное псевдосостояние (initial pseudo state) — вершина графа конечного автомата, которая по умолчанию является состоянием-источником для начального перехода в моделируемом поведении.

Несовместимое подсостояние (disjoint substate) — подсостояние, в котором подсистема не может находиться одновременно с другими подсостояниями одного и того же композитного состояния.

Нить <управления> (thread <of control>) — некоторый простой путь выполнения программы, динамической модели или любого другого потока управления.

Нотация (notation) — система условных обозначений, принятая в некотором языке для изображения и визуализации модели.

Обобщение (generalization) — таксономическое отношение между общим классификатором (родителем или предком) и специальным классификатором (дочерним или потомком).

Обработчик исключения (exception handler) — спецификация деятельности, выполнение которой происходит в случае исключения в ходе выполнения некоторого защищенного узла.

Объект (object) — отдельный экземпляр класса, который создается на этапе реализации модели или выполнения программы.

Ограничение (constraint) — логическое условие, выраженное текстом на естественном языке или на языке, который может быть прочитан некоторой машиной, с целью декларирования семантики некоторого элемента модели. Ограничения являются одним из трех механизмов расширения языка UML 2.0.

Ограничение взаимодействия (interaction constraint) — логическое выражение, которое выступает в роли сторожевого условия некоторого операнда в комбинированном фрагменте.

Ограничение на продолжительность (duration constraint) — ограничение, которое ссылается на некоторый интервал продолжительности.

Операнд взаимодействия (interaction operand) — отдельный фрагмент взаимодействия, предназначенный для использования в качестве внутренней части комбинированного фрагмента.

Оператор взаимодействия (interaction operator) — ключевое слово, которое определяет тип комбинированного фрагмента. Является перечислением следующих литералов: `alt`, `assert`, `break`, `critical`, `ignore`, `consider`, `loop`, `neg`, `opt`, `par`, `seq`, `strict`.

Операция (operation) — характеристика поведения, которая является общей для всех объектов данного класса. Представляет некоторый сервис, который может быть востребован одним объектом у другого.

Организационная единица (organization unit) — пакет, в состав которого могут входить сотрудники, бизнес-сущности, бизнес-варианты использования, отдельные диаграммы языка UML 2.0 и другие организационные единицы. Синоним — организационное подразделение.

Ортогональное композитное состояние (orthogonal composite state) — композитное состояние, содержащее более одного региона.

Отношение (relationship) — произвольная семантическая взаимосвязь между отдельными элементами модели. Примерами отношений в языке UML 2.0 являются ассоциации и обобщения.

Отправитель, **объект-отправитель** (sender <object>) — объект, который отправляет некоторую информацию объекту-получателю.

Пакет (package) — элемент модели, используемый для группировки других элементов модели.

Параллельное подсостояние (concurrent substate) — подсостояние, в котором подсистема может находиться одновременно с другими параллельными подсостояниями, входящими в одно составное состояние.

Параллельность (concurrency) — возможность одновременного выполнения двух и более видов деятельности в течение некоторого интервала времени.

Параметр (parameter) — спецификация аргумента, который используется при выполнении операции или при вызове характеристики поведения.

Параметризуемый элемент, **шаблон** (parameterized element, template) — описатель класса, который содержит один или несколько неопределенных параметров.

Пассивный класс (passive class) — класс, каждый экземпляр которого выполняется в контексте некоторого другого объекта.

Паттерн (pattern) — параметризованная кооперация вместе с описанием базовых принципов ее использования в форме параметров, которые ограничивают использование объектов в контексте ролей элементов рассматриваемого паттерна.

Передача <сообщения> (send <message>) — прохождение информации от отправителя к получателю.

Переход (transition) — направленное отношение между двумя состояниями, первое из которых является вершиной источником (source vertex), а второе — целевой вершиной (target vertex).

Перечисление (enumeration) — список именованных значений, используемых в качестве области значений атрибута отдельного типа. Например, Цвет = {красный, зеленый, голубой}.

Поведение (behavior) — наблюдаемый результат выполнения операции или наступления некоторого события. В контексте конечного автомата является спецификацией того, как экземпляр классификатора изменяет значения отдельных характеристик в течение своего времени жизни.

Подкласс (subclass) — класс, который является составной частью другого класса.

Подпакет (subpackage) — пакет, который является составной частью другого пакета.

Подсистема (subsystem) — совокупность модельных элементов, которая представляет фрагмент поведения физической системы. Подсистема представляется своими интерфейсами и имеет операции. Модельные элементы подсистемы могут быть разделены на элементы спецификации и элементы реализации.

Подсостояние (substate) — состояние, которое заключено в регион композитного состояния.

Подтип (subtype) — классификатор, который является специализацией более общего классификатора типа в отношении обобщения.

Полиморфизм (polymorphism) — свойство одноименных методов или операций выполнять различные действия или обладать различным поведением в зависимости от того, к какому из классов они относятся.

Получатель <объект-получатель> (receiver <object>) — объект, который наделен способностью воспринимать информацию, передаваемую другими объектами.

Получать <сообщение> (receive <a message>) — способность одного элемента модели воспринимать информацию от других элементов.

Помеченное значение (tagged value) — явное определение свойства как пары "имя — значение". В помеченном значении само имя называют *тегом* (tag). Отдельные теги предопределены в языке UML 2.0, другие — могут быть определены пользователем. Помеченные значения являются одним из трех механизмов расширения языка UML 2.0.

Порт (port) — свойство классификатора, которое специфицирует отдельную точку взаимодействия между этим классификатором и его окружением или между классификатором и его внутренними частями.

Поставщик (supplier) — классификатор, который предоставляет некоторые сервисы другим классификаторам-клиентам. Часто называется также сервером.

Постусловие (postcondition) — ограничение, которое должно быть истинным после завершения выполнения операции.

Поток объектов (object flow) — дуга деятельности, по которой передаются только маркеры объектов или данных.

Поток управления (control flow) — дуга деятельности, которая связывает между собой два узла деятельности.

Потомок (child) — специализация одного из элементов отношения обобщения, называемого в этом случае родителем. Примерами потомков являются подкласс, подсистема.

Предметная область (domain) — область знаний или некоторой деятельности, характеризуемая множеством понятий и терминологией, понятной специалистам в этой предметной области.

Предоставляемый интерфейс (provided interface) — интерфейс, который компонент предлагает для своего окружения.

Представление (view) — проекция модели, которая рассматривается с определенной точки зрения. В отдельном представлении учитываются только существенные аспекты модели, а все второстепенные детали могут быть опущены.

Предусловие (precondition) — ограничение, которое должно быть истинным перед началом выполнения операции.

Применение кооперации (collaboration use) — описание реализации кооперации в форме множества взаимодействующих элементов посредством связывания этих элементов с ролями данной кооперации.

Примечание (note) — элемент графической нотации, присоединенный к другому элементу модели и не имеющей собственной семантики.

Продолжение (continuation) — специальная метка, которая позволяет разбивать операнды комбинированного фрагмента Альтернативы на две и более части и комбинировать их траектории в различных фреймах.

Продолжительность (duration) — временное расстояние между двумя временными выражениями, которые соответствуют двум моментам времени.

Проектирование (design) — часть процесса разработки программной системы, основная цель которой — решить, как система должна быть реализована, чтобы удовлетворять предъявляемым к ней требованиям.

Проекция (projection) — отображение некоторого множества в некоторое его подмножество.

Проекция представления (view projection) — некоторая проекция элементов модели на элементы представления. Проекция представления обеспечивает расположение и стиль каждому элементу представления.

Производный элемент (derived element) — элемент модели, который может быть вычислен посредством другого элемента, но, тем не менее, включается в модель для ясности или удобства разработки, хотя при этом в модель не вносится никакой новой семантики.

Простейший тип (primitive type) — предопределенный в языке UML 2.0 тип данных, не содержащий никакой вложенной структуры. Это следующие типы: `Integer`, `Boolean`, `String` и `UnlimitedNatural`.

Простое наследование (single inheritance) — семантический вариант обобщения, при котором тип может иметь только один супертип.

Простое состояние (simple state) — состояние, которое не имеет внутренних регионов и подсостояний.

Пространство имен (namespace) — часть модели, в которой определяются и используются имена элементов модели. В одном пространстве имен каждое имя должно быть уникальным (единственным).

Протокольное состояние (protocol state) — состояние протокольного конечного автомата, которое представляет некоторую открытую для наблюдения стабильную ситуацию контекста его классификатора.

Протокольный конечный автомат (protocol state machine) — конечный автомат, предназначенный для спецификации допустимых состояний и условий при вызове операций классификатора.

Протокольный переход (protocol transition) — переход, который специфицирует некоторый законный переход для операции, представленной в форме протокольного конечного автомата.

Процесс (process) — самостоятельная единица параллельности и исполнения в операционной системе. В контексте ООАП процесс относится к разработке программной системы и представляет собой последовательность взаимосвязанных этапов, направленных на реализацию проекта системы.

Процесс разработки (development process) — множество частично упорядоченных шагов, каждый из которых направлен на решение отдельной задачи в ходе процесса конструирования и реализации моделей.

Псевдосостояние (pseudo state) — абстрактный элемент модели, который включает в себя различные типы вспомогательных вершин в графе конечного автомата.

Псевдосостояние выбора (choice pseudo state) — псевдосостояние, предназначенное для моделирования нескольких альтернативных ветвей при реализации поведения конечного автомата.

Псевдосостояние глубокой истории (deep pseudo state) — псевдосостояние, предназначенное для представления последней активной конфигурации композитного состояния после выхода из него.

Псевдосостояние неглубокой истории (shallow pseudo state) — псевдосостояние, предназначенное для представления самого последнего активного подсостояния композитного состояния после выхода из него.

Псевдосостояние соединения (junction pseudo state) — вершина со свободной семантикой, которая используется для соединения вместе нескольких переходов в графе конечного автомата.

Путь коммуникации (communication path) — ассоциация между двумя целями развертывания, посредством которой они обладают способностью обмениваться сигналами и сообщениями.

Разбиение (partition) — элемент модели, предназначенный для группировки действий, которые относятся к одной деятельности и имеют некоторую общую характеристику. В языке UML 1.*x* разбиение называлось дорожкой (swimlane).

Развертывание (deployment) — размещение артефакта или экземпляра артефакта на некоторой цели развертывания.

Размещаемый артефакт (deployed artifact) — абстрактный метакласс для представления артефакта или экземпляра артефакта, который должен быть размещен на цели развертывания.

Расширение (extend) — отношение зависимости, которое специфицирует тот факт, что один вариант использования задействует функциональность или поведение другого варианта использования не всегда, а только при выполнении некоторых дополнительных условий.

Реализация (realization) — специализированное отношение зависимости между двумя элементами модели, один из которых представляет некоторую спецификацию (поставщик), а другой представляет его реализацию (клиент).

Регион (region) — специальный элемент модели, который содержит состояния и переходы и является частью композитного состояния или конечного автомата.

Регион прерываемой деятельности (interruptible activity region) — элемент модели, который поддерживает прекращение всех потоков маркеров в некоторой деятельности.

Родитель, предок (parent) — в отношении обобщения общий элемент или суперкласс.

Роль (role) — имеющее имя специфическое поведение некоторой сущности, рассматриваемой в определенном контексте. Роль может быть статической или динамической.

Свойство (property) — именованное значение, которое обозначает характеристику некоторого элемента. Свойство имеет собственную семантику. Некоторые свойства предопределены в языке UML 2.0, другие могут быть определены разработчиком.

Связь (link) — экземпляр произвольной ассоциации, которая обеспечивает канал для направленной передачи сообщений между линиями жизни.

Семантика (semantics) — система правил и соглашений, определяющих толкование и придание смысла конструкциям некоторого языка.

Сигнал (signal) — спецификация асинхронной коммуникации между линиями жизни. Сигналы могут иметь параметры.

Сигнатура (signature) — имя и параметры некоторого свойства поведения. Сигнатура может включать необязательный возвращаемый параметр.

Система (system) — в метамодели языка UML 2.0 подсистема самого верхнего уровня абстракции.

Слияние пакетов (package merge) — направленное отношение между двумя пакетами, один из которых расширяет свое содержание посредством добавления содержимого другого пакета.

Слот (slot) — элемент модели, предназначенный для представления того, что сущность, моделируемая посредством спецификации экземпляра, имеет конкретное значение или значения для некоторой своей структурной характеристики.

Собирающий соединитель (assembly connector) — соединитель, который связывает два компонента в контексте предоставляемых и требуемых сервисов.

Событие (event) — спецификация некоторых условий, которые оказывают влияние на поведение моделируемой сущности.

Соединитель (connector) — отношение, которое обеспечивает взаимосвязь или коммуникацию между двумя или более экземплярами классификаторов, в частном случае — экземплярами классов.

Сообщение (message) — элемент модели, предназначенный для представления отдельной коммуникации между линиями жизни некоторого взаимодействия.

Составное состояние (composite state) — состояние, имеющее в качестве внутренних подсостояний параллельные или последовательные состояния.

Составной переход (compound transition) — производное семантическое понятие, которое представляет "семантически полный" путь, совершаемый одним или несколькими переходами.

Состояние (state) — условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет некоторому условию, выполняет некоторую деятельность или ожидает некоторого события.

Состояние действия (action state) — состояние, которое представляет выполнение некоторого атомарного действия, такого как обращение к операции.

Состояние потока объекта (object flow state) — состояние в графе деятельности, которое представляет переход объекта от выполнения действий выхода из одного состояния к выполнению входных действий в другом состоянии.

Сотрудник (business worker) — индивидуум, который действует внутри моделируемой системы, такой как компания или организация, взаимодействует с другими сотрудниками и манипулирует бизнес-сущностями в процессе выполнения отдельных операций бизнес-процесса.

Спецификация (specification) — декларативное описание некоторой сущности или поведения.

Спецификация развертывания (deployment specification) — спецификация множества свойств, которые определяют параметры выполнения артефакта компонента, развертываемого на некотором узле.

Спецификация экземпляра (instance specification) — элемент модели, который представляет описание произвольной сущности типа классификатора.

Срабатывание <перехода> (fire) — выполнение перехода из одного состояния в другое состояние.

Среда выполнения (execution environment) — узел, который обладает функциональностью, необходимой для практического выполнения развернутых на нем исполнимых артефактов.

Статическая классификация (static classification) — семантический вариант обобщения, при которой объект не может изменять классификатор.

Стереотип (stereotype) — элемент модели, который расширяет семантику базового элемента метамодели языка UML 2.0 и основывается на уже существующих и описанных в метамодели языка UML 2.0 типах или классах. Является одним из трех механизмов расширения в языке UML 2.0.

Сторожевое условие (guard condition) — условие, которое должно быть обязательно выполнено для того, чтобы сработал ассоциированный с ним переход. В языке UML 2.0 часто используется в сокращенной форме — guard.

Строка (string) — последовательность символов текста. Набор конкретных допустимых символов зависит от реализации и, как правило, включает в себя интернациональную кодировку алфавита и символов графики.

Структурная модель (structured model) — модель, предназначенная для описания статической структуры сущностей или элементов некоторой системы, такие как классы, интерфейсы, атрибуты и отношения.

Структурная характеристика (structural feature) — типизированная характеристика классификатора, которая специфицирует структуру его экземпляров. Структурной характеристикой класса является атрибут.

Субъект (subject) — любой элемент модели, который обладает функциональным поведением. В контексте диаграммы вариантов использования субъект представляется в форме границы системы (system boundary), разделяющей элементы модели, которые принадлежат проектируемой системе, от любых других элементов предметной области или окружения системы.

Суперкласс (superclass) — общий класс в отношении обобщения.

Супертип (supertype) — общий классификатор типа в отношении обобщения.

Сценарий (scenario) — текст с описанием последовательности действий, выполняемых сущностями в контексте моделирования некоторого поведения. В контексте языка UML 2.0 сценарий используется для дополнительной спецификации взаимодействия актеров и вариантов использования, не исключая соответствующей графической нотации.

Точка входа (entry point) — псевдосостояние, предназначенное для моделирования входа в некоторый конечный автомат или композитное состояние.

Точка выхода (exit point) — псевдосостояние, предназначенное для моделирования выхода из некоторого конечного автомата или композитного состояния.

Тип (type) — стереотип класса, который используется для спецификации области значений экземпляров (объектов) совместно с операциями, применимыми к этим объектам.

Тип данных (datatype) — описатель множества значений, которые могут принимать атрибуты классов и возвращать в качестве значений операции классов.

Требование (requirement) — желательное свойство, характеристика или поведение системы.

Требуемый интерфейс (required interface) — интерфейс, который необходим компоненту от своего окружения для выполнения заявленной функциональности, контракта или поведения.

Триггер (trigger) — отношение события с поведением, которое может оказывать влияние на экземпляр классификатора. В нотации языка UML 2.0 обычно представляется в форме перехода с именем события.

Узел (node) — классификатор, который представляет собой некоторый вычислительный ресурс для развертывания на нем различных артефактов.

Узел деятельности (activity node) — абстрактный класс для отдельных точек в потоке деятельности, соединенных дугами.

Узел завершения (terminate node) — псевдосостояние, вход в который означает завершение выполнения поведения конечного автомата в контексте его объекта.

Узел объекта (object node) — узел абстрактной деятельности, который является частью определяющего потока объектов в деятельности.

Узел параметра деятельности (activity parameter node) — узел объекта для моделирования входов и выходов деятельности.

Узел разделения (fork node) — узел управления, который расщепляет поток на несколько параллельных потоков.

Узел решения (decision node) — узел управления, который выбирает между выходящими потоками.

Узел слияния (merge node) — узел управления, который соединяет вместе несколько альтернативных потоков.

Узел соединения (join node) — узел управления, который синхронизирует несколько потоков.

Узел управления (control node) — абстрактный узел деятельности, который предназначен для координации потоков в деятельности.

Узел финала деятельности (activity final node) — узел управления, который прекращает или останавливает все потоки в деятельности.

Узел финала потока (flow final node) — узел деятельности, который завершает отдельный поток управления или поток объектов, не завершая содержащей его деятельности.

Управляющий класс (control class) — класс, отвечающий за координацию действий других классов. На каждой диаграмме классов должен быть хотя бы один управляющий класс, причем объектам управляющего класса посылают мало сообщений, а он рассылает много сообщений.

Устройство (device) — узел, который обладает некоторым общим вычислительным ресурсом со способностью обрабатывать развернутые на нем артефакты.

Уровень (layer) — организация классификаторов или пакетов на одном уровне абстракции. Уровень представляет горизонтальный срез архитектуры модели, в то время как декомпозиция представляет ее вертикальный срез.

Физическая система (physical system) — реально существующий прототип модели системы. В контексте языка UML 2.0 обозначает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

Финальное состояние (final state) — специальный вид состояния, предназначенный для моделирования завершения конечного автомата или региона, в котором оно содержится.

Фрагмент взаимодействия (interaction fragment) — абстрактное понятие для представления общей единицы или части взаимодействия.

Характеристика (feature) — элемент модели, предназначенный для спецификации отдельных аспектов структуры и поведения экземпляров классификаторов. Характеристиками класса являются операция и атрибут.

Характеристика поведения (behavioral feature) — характеристика классификатора, которая специфицирует некоторый аспект поведения его экземпляров. Характеристикой поведения класса является операция или метод.

Хранилище данных (data store) — разновидность центрального буфера для постоянного хранения объектов или другой информации.

Центральный буфер (central buffer) — узел объекта для управления потоками из нескольких источников и мест назначения.

Цель развертывания (deployment target) — абстрактный метакласс для указания местоположения размещаемого артефакта.

Часть (part) — свойство, которое является элементом внутренней структуры композитного классификатора, в частном случае — класса.

Шаблон (template) — классификатор, который в своем описании имеет несколько формальных параметров.

Шлюз (gate) — точка соединения для установления взаимосвязи сообщения за пределами фрагмента взаимодействия с сообщением внутри этого фрагмента взаимодействия.

Экземпляр (instance) — элемент модели, который является конкретным представителем классификатора.

Элемент (element) — атомарная составляющая представления модели.

Элемент модели (model element) — элемент, который является некоторой абстракцией, извлеченной из моделируемой системы.

Элемент представления (view element) — текстовая и/или графическая проекция некоторой совокупности элементов модели.

Этап анализа (analysis time) — часть процесса ООАП, относящаяся к выполнению фазы анализа процесса разработки программной системы.

В заключение приводится список встречающихся в книге общепринятых сокращений с кратким пояснением их содержательного смысла.

ALM (Application Lifecycle Management) — управление жизненным циклом приложений. Методология разработки программных проектов компании Borland, основанная на интеграции средств поддержки всех этапов выполнения проектов в единой инструментальной среде.

BPML (Business Process Modeling Language) — язык моделирования бизнес-процессов. Специальный язык и графическая нотация, ориентированная на моделирование бизнес-процессов. Оказал непосредственное влияние на внесенные дополнения в диаграммы деятельности языка UML 2.0.

CASE (Computer Aided Software Engineering) — автоматизированная разработка программного обеспечения. Методология разработки программ, основанная на комплексном использовании компьютеров не только для написания исходных кодов, но и для анализа и моделирования соответствующей предметной области.

CBD (Component-Based Development) — компонентная разработка приложений. Новая парадигма разработки программ, интенсивно развиваемая в последнее время. Основные идеи этой парадигмы во многом созвучны с технологией CORBA.

CORBA (Common Object Request Broker Architecture) — общая архитектура брокеров объектных запросов. Этим термином обозначают технологию, архитектуру и набор стандартов для создания распределенных программных приложений. Является главным стратегическим направлением деятельности консорциума OMG. Оценивается специалистами как новый многообещающий подход к построению программных систем, который предполагает использование готовых компонентов и предлагает технологию их интеграции.

DFD (Data Flow Diagram) — диаграмма потоков данных. Графическая нотация, ориентированная на создание моделей распределенной обработки данных.

ERD (Entity-Relation Diagram) — диаграмма "сущность — связь". Графическая нотация, предложенная П. Ченом и получившая дальнейшее развитие в работах Р. Баркера.

IDL (Interface Definition Language) — язык определения интерфейса. Специальный декларативный язык для описания стандартов CORBA, для которого существуют компиляторы в большинство современных языков программирования.

MDA (Model Driven Architecture) — управляемая моделями архитектура. Использование моделей для управления всеми аспектами архитектуры системы и программных приложений.

OCL (Object Constraint Language) — язык объектных ограничений. Составная часть унифицированного языка моделирования UML 2.0. Служит для записи ограничений и правил правильного построения выражений UML 2.0.

OMG (Object Management Group) — название консорциума, созданного в 1989 году для разработки индустриальных стандартов с их последующим использованием в процессе создания интероперабельных неоднородных распределенных объектных сред.

OMT (Object Modeling Technique) — методология и графическая нотация объектно-ориентированного анализа и проектирования, разработчиком которых является Джеймс Румбах (James Rumbaugh) и которая реализована в CASE-средстве Rational Rose 98.

OOA/OOD, OOA&D (Object-Oriented Analysis/Design) — объектно-ориентированный анализ и проектирование (ООАП). Методология разработки программных систем, в основу которой положена объектно-ориентированная концепция представления моделей предметной области в форме классов, обладающих структурными свойствами и поведением.

OOP (Object-Oriented Programming) — объектно-ориентированное программирование (ООП). Совокупность принципов, технологии и инструментальных средств для создания программных систем, в основу которых закладывается архитектура взаимодействия объектов. Говоря об ООП, чаще всего отмечают его основные принципы: абстракция, наследование, полиморфизм и инкапсуляция.

OOSE (Object-Oriented Software Engineering) — методология и элементы графической нотации объектно-ориентированного анализа и проектирования,

одним из основных разработчиков которых является Айвар Джекобсон (Ivar Jacobson).

PIM (Platform Independent Model) — платформенно-независимая модель. Модельное представление системы, которое концентрирует внимание на общей архитектуре системы и скрывает детали, необходимые для ее реализации на отдельной платформе.

PSM (Platform Specific Model) — платформенно-зависимая модель. Модельное представление системы, которое специфицирует все особенности, необходимые для ее реализации на конкретной платформе.

RUP (Rational Unified Process) — рациональный унифицированный процесс, совокупность понятий и принципов, позволяющих организовать процесс ООАП на основе некоторой методики, которая использует отдельные элементы нотации языка UML 2.0.

SADT (Structured Analysis and Design Technique) — технология структурного анализа и проектирования, разработчиком которой является Дуглас Росс (D. Ross).

UML (Unified Modeling Language) — унифицированный язык моделирования для описания, визуализации и документирования объектно-ориентированных систем в процессе их разработки.

XMI (XML-based Model Interchange) — обмен моделей на базе XML. Механизм преобразования данных о моделях с использованием формата XML. Информация о моделях сохраняется в файле формата XML, который может быть импортирован другими инструментальными средствами.

Литература

1. Амриш К.И., Ахмед Х. З. Разработка корпоративных Java-приложений с использованием J2EE и UML: Пер. с англ. — М.: Вильямс, 2002.— 272 с.
2. Боггс У., Боггс М. UML и Rational Rose: Пер. с англ. — М.: ЛОРИ, 2000. — 582 с.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++: Пер. с англ. — М.: Бином; СПб.: Невский диалект, 1999. — 560 с.
4. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя: Пер. с англ. — М.: ДМК, 2000. — 432 с.
5. Буч Г., Якобсон А., Рамбо Дж. UML. Классика CS. 2-е изд. — СПб.: Питер, 2006. — 736 с.
6. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования: Пер. с англ. — СПб.: Питер, 2001.— 368 с.
7. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений: Пер. с англ. — М.: ДМК Пресс, 2002. — 704 с.
8. Грехем И. Объектно-ориентированные методы. Принципы и практика: Пер. с англ. — М.: Вильямс, 2004. — 880 с.
9. Грибачев К. Г. Delphi и Model Driven Architecture. Разработка приложений баз данных. — СПб.: Питер, 2004. — 348 с.
10. Коналлен Дж. Разработка Web-приложений с использованием UML. Пер. с англ. — М.: Вильямс, 2001. — 288 с.
11. Ларман К. Применение UML и шаблонов проектирования: Пер. с англ. — М.: Вильямс, 2001. — 496 с.

12. Ларман К. Применение UML и шаблонов проектирования. 2-е издание: Пер. с англ. — М.: Вильямс, 2002. — 624 с.

13. Леоненков А. В. Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose. — М.: Интернет-Университет Информационных технологий: БИНОМ, Лаборатория знаний, 2006. — 320 с.

14. Леоненков А. В. Самоучитель UML. 2-е издание. — СПб.: БХВ-Петербург, 2004. — 432 с.

15. Леоненков А. В. Самоучитель UML. — СПб.: БХВ-Петербург, 2001. — 304 с.

16. Леффингуэлл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход: Пер. с англ. — М.: Вильямс, 2002. — 448 с.

17. Нейбург Э. Дж., Максимчук Р. А. Проектирование баз данных с помощью UML: Пер. с англ. — М.: Вильямс, 2002. — 288 с.

18. Рамбо Дж., Якобсон А., Буч Г. UML: специальный справочник: Пер. с англ. — СПб.: Питер, 2001. — 656 с.

19. Розенберг Д., Скотт К. Применение объектного моделирования с использованием UML и анализ прецедентов: Пер. с англ. — М.: ДМК Пресс, 2002. — 160 с.

20. Санблэд С. Разработка масштабируемых приложений для Microsoft Windows. Мастер-класс. — М.: ИТД "Русская редакция", 2002. — 416 с.

21. Фаулер М., Скотт К. UML. Основы: Пер. с англ. — СПб.: Символ-Плюс, 2002. — 192 с.

22. Шаллоуей А., Тротт Дж. Р. Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию: Пер. с англ. — М.: Вильямс, 2002. — 288 с.

23. Шмуллер Д. Освой самостоятельно UML за 24 часа: Пер. с англ. — М.: Вильямс, 2002. — 352 с.

24. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения: Пер. с англ. — СПб.: Питер, 2002. — 496 с.

25. Cockburn A. Writing Effective Use Cases. Addison-Wesley, 2000.

26. Grand M. Patterns in Java. A Catalog of Reusable Design Patterns Illustrated with UML. — Wiley & Sons, 1998. Vol. 1 — 468 p. Vol. 2 — 356 p.

27. Jacobson I., Booch G., Rumbaugh J. Unified Software Development Process. Addison-Wesley, 1999. — 512 p.
28. Kruchten P. The Rational Unified Process. An Intriduction. Addison-Wesley, 2000. — 300 p.
29. Rumbaugh J., Jacobson I., Booch G., The Unified Modeling Language. Reference Manual. Addison-Wesley, 1999. — 550 p.
30. Starr L. Executable UML. How to build class models. — Prentice Hall PTR, 2002. — 418 p.
31. Wampler B.E. The Essence Object-Oriented Programming with Java and UML. — Addison-Wesley, 2002. — 290 p.

Предметный указатель

A

Abstract class 111
Abstract operation 120
Abstraction 16
Accept event action 263
Accept time event action 264
Action 246
Active class 109
Activity 246
Activity diagram 58
Activity edge 248
Activity final node 255
Activity node 247
Activity parameter node 271
Activity partition 275
Actor 79
Aggregation 140
Artifact 389
Assembly connector 368
Association 84, 124
Association class 131
Association end 124
Association name 124
Attribute 112

B

Behavior 309
Behavior port 162
Behavioral feature 106
Behavioral models 39
Binary association 124

C

CASE-tools 14
CASE-средство 14
Central buffer 267
Choice pseudo state 319
Class 16, 107
Class diagram 58, 104
Class name 110
Classifier 105
Collaboration 165
Collaboration role 166
Collaboration use 167
Combined fragment 213
Comment 82
Communication diagram 58, 286
Communication path 396
Component 360
Component diagram 59, 357
Composite state 339
Composite structure diagram 58
Composition 142
Compound transition 332
Computer Aided Software Engineering (CASE) 14
Connector 158
Connector end 159
Constraint 67

Continuation 232
Control flow 249
Control node 254

D

Data Flow Diagrams (DFD) 20
Data store 268
Decision node 256
Deep pseudo state 327
Delegation connector 370
Dependency 86, 144
Deployed artifact 388
Deployment 393
Deployment diagram 59, 383
Deployment specification 390
Deployment target 388
Derive attribute 112
Device 387
Direct substate 340
Duration 239
Duration constraint 241
Duration interval 240
Duration observation action 240

E

Element import 180
Encapsulation 18
Entity-Relationship Diagrams (ERD) 20
Entry point 324
Event 311
Exception handler 280
Execution environment 385
Execution specification 206
Exit point 324
Extend 88
Extension point 88

F

Feature 106
Final state 329
Flow final node 256
Fork node 259
Fork vertex 322

G

Gate 234
Generalization 134
Generalization relationship 92
Generalization set 137
Guard condition 330

H

Handler body 280

I

Include 86
Indirect substate 340
Inheritance 16
Initial node 254
Initial pseudo state 318
Input pin 269
Instance specification 195, 390
Instance value 199
Interaction 203
Interaction constraint 215
Interaction fragment 213
Interaction operand 214
Interaction operator 216
Interaction overview
diagram 58, 298
Interaction use 226
Interface 145
Internal structure 154
Interruptible activity region 278
Interrupting edge 278
Interval 238

J

Join node 260
Join vertex 323
Junction pseudo state 321

L

Least common ancestor (LCA) 336
Lifeline 205
Link 158, 289

M

Manifestation 395
Merge node 258
Message 208
Message kind 209
Message sort 208
Methodology 13
Model 10
Model Driven Architecture (MDA) 22
Multiplicity 116

N

N-ary association 130
Node 384
Notation 12, 38
Note 82

O

Object 194
Object Constraint Language (OCL) 52, 501
Object diagram 58, 194
Object flow 249
Object node 265
Object-Oriented Analysis/Design 15
Operation 117
Orthogonal regions 340
Output pin 269

P

Package 176
Package diagram 58, 175
Package import 179
Package merge 183
Parameter 120
Parameter set 273
Part 156
Part decomposition 228
Passive class 109
Platform Independent Model (PIM) 22
Platform Specific Model (PSM) 23
Polymorphism 19
Port 161
Postcondition 122
Precondition 122
Profiles 359
Property 155
Protected node 280
Protocol state 351
Protocol state machine 350
Protocol transition 352
Provided interface 146, 163, 364
Pseudo state 317

Q

Qualified name 111
Qualifier 133

R

Realization 145, 375
Region 339
Relationship 83
Required interface 146, 163, 364
Requirement 97
Role 124

S

Scenario 99
Semantics 38
Send signal action 262
Sequence diagram 58, 202
Shallow pseudo state 325
Signal 211
Signal event 212
Simple state 314
Slot 197
Source vertex 310
State 310
State invariant 230
State machine 309
State machine diagram 58, 308
Stereotype 66
Structural feature 106
Structured Analysis and Design Technique (SADT) 20
Structured models 39

Subject 75
Submachine state 347
Substate 340

T

Tagged value 69
Target vertex 310
Template 148
Terminate node 319
Time constraint 239
Time event 236
Time expression 236
Time interval 238
Time observation action 237
Timing diagram 58, 301
Together Designer 404
Token 251
Transition 310
Trigger 312

U

Unified Modeling Language (UML) 13
Use case 77
Use case diagram 57, 73

V, X

View 10
Visibility 112
Xor-association 128

А

Абстрактный синтаксис 51
Абстракция 16
Агрегация 140
  композитная 142
Актер 79, 436
  второстепенный 85
  главный 85
Артефакт 358, 389
  размещаемый 388
Архитектура, управляемая моделями 22
Ассоциация 84, 124
  N-арная 130
  бинарная 124
  имя 124
  исключающая 128
  конец 124
  рефлексивная 124
Ассоциация-класс 131
Атрибут 112
  вычислимый 112
  производный 112

Б

Белый ящик 361, 364

В

Вариант использования 77
Вершина:
  выбора 319
  глубокой истории 327
  источник 310
  начальная 318
  неглубокой истории 325
  разделения 322
  слияния 323
  соединения 321
  целевая 310
Взаимодействие 203
Вид:
  видимости 114
  сообщения 209
Видимость 112
Временное выражение 236
Временное ограничение 239
Временное событие 236
Выбор 319
Выполнение до завершения 312
  шаг 312
Выражение поведения 330

Г

Глубокая история 327
Графический узел 60

Д

Действие 246
  на входе 315
  на переходе 334
  наблюдения:
    времени 237
    продолжительности 240
  ожидания временного события 264
  передачи сигнала 262
  приема события 263
Декомпозиция части 228
Деятельность 246
Диаграмма 20, 55
  вариантов использования 57, 73, 431
    пример 94, 100
  временная 58, 285, 301
    пример 304—306
    формы 304
  деятельности 58, 245, 474
  классов 58, 104, 444
    пример 150
  коммуникации 58, 285, 286, 468
  композитной структуры 58, 154, 455
  компонентов 59, 357, 484
  конечного автомата 58, 308, 479
  обзора взаимодействия 58, 285, 298
  объектов 58, 194
  пакетов 58, 175
  поведения 56
  последовательности 58, 202, 460
  потоков данных 21
  развертывания 59, 383, 488
  структуры 56
  сущность — связь 20
  функционального моделирования 21
Дуга:
  деятельности 248
  прерывающая 278

З

Зависимость 144
Значение экземпляра 199

И

Импорт:
  пакета 179
  элемента 180
Имя класса 110
  квалифицированное 111
  простое 111
Инвариант состояния 230
Инкапсуляция 18
Интервал 238
  временной 238
  продолжительности 240
Интерфейс 145, 364
  предоставляемый 146, 163, 364
  требуемый 146, 163, 364
Использование взаимодействия 226

К

Квалификатор 133
Класс 16, 107
  абстрактный 111
  активный 109
  композитный 156
  пассивный 109
Классификатор 105
  композитный 155
Комбинированный фрагмент 213
Комментарий 82
Композит 142
Композиция 142
Компонент 356, 358, 360, 376
  экземпляр 363
Конец соединителя 159
Конечный автомат 309
  контейнерный 348
  протокольный 350
Контакт:
  входной 269
  выходной 269
Конфигурация состояний 341
Конфликтующие переходы 337
Кооперация 165
  применение 167
Кратность 116

Л

Линия жизни 205, 288

М

Манифестация 395
Маркер 251
Метакласс 45
Мета-метамодель 39
Метамодель 40
Метасистема 11
Методология 13
Механизм расширения 65
Множество:
  обобщения 137
  параметров 273
Моделирование 10
  физическое 356
Модель 10, 40
  вариантов использования 98
  коммуникации 295
  платформенно-зависимая 23
  платформенно-независимая 22
  поведения 39
  структурная 39
Модель FURPS+ 97

Н

Наследование 16
  множественное 136
Начальный узел 254
Неглубокая история 325
Нотация 12, 38

О

Обобщение 134
Обработчик исключения 280
Объект 194
  сирота 195
Объектно-ориентированный анализ и проектирование 15
Ограничение 67
  взаимодействия 215
  на продолжительность 241
Операнд взаимодействия 214
Оператор взаимодействия 216
Операция класса 117
  абстрактная 120
  переопределенная 122
    согласованная 123
  сигнатура 118
Отношение 83
  включения 86
  зависимости 86, 373
  обобщения 92
  расширения 88

П

Пакет 42, 176
Параметр 120
Передача сигнала 333
Переход 308, 310, 329
  верхнеуровневый 343
  внешний 343
  внутренний 316
  завершения 331
  начальный 318
  протокольный 352
  рефлексивный 332
  составной 332
Поведение 309
  входное 315
  выходное 316
Подсистема 11
Подсостояние 340
  непрямое 340
  прямое 340
Полиморфизм 19
Помеченное значение 69
Порт 161, 366
  конечный 367
  передающий 367
  поведения 162
Постусловие 122
Поток:
  объектов 249
  управления 249
Правила корректного построения выражений 52

Правило:
входа по умолчанию 341
для ортогональных регионов 344
выхода по умолчанию 342
для ортогональных регионов 344
Представление 10
Предусловие 122
Прием:
сигнала 333
триггера 333
Примечание 82
Продолжение 232
Продолжительность 239
Профиль 359
Псевдосостояние 308, 317
выбора 319
глубокой истории 327
начальное 318
неглубокой истории 325
соединения 321
Путь коммуникации 396

Р

Разбиение деятельности 275
Развертывание 393
Разделение 322
Реализация 145, 375
Регион 339
прерываемой деятельности 278
Регионы ортогональные 340
Роль:
класса 124
кооперации 166

С

Свойство 155
Связь 158, 289
Семантика 38, 52
Сигнал 211, 291
Система:
переход 11
сложная 11
структура 10
функционирование 11
Слияние пакетов 183
Слот 197
Событие 311
завершения 331
отложенное 317
сигнала 212
Соглашение об именах в UML 2.0 42
Соединение 321
Соединитель 158, 368
делегирующий 370
собирающий 368
Сообщение 208, 290
синхронное 291
создания объекта 209
Сорт сообщения 208
Состояние 310
анонимное 315
композитное 339
наименьший общий предок 336
подавтомата 347
простое 314
протокольное 351
финальное 329
Спецификация:
выполнения 206
развертывания 390
экземпляра 195, 390
Среда выполнения 385
Стереотип 66
<<access>> 179, 181
<<actor>> 80
<<artifact>> 389
<<bind>> 150, 171
<<centralBuffer>> 268
<<component>> 360
<<create>> 319
<<datastore>> 269
<<delegate>> 372
<<deploy>> 394
<<deployment spec>> 391
<<device>> 387
<<executable>>. 390
<<executionEnvironment>> 386
<<extend>> 88
<<external>> 277
<<implement>> 375
(окончание рубрики см. на стр. 558)

Стереотип *(окончание)*:
<<import>> 179, 181
<<include>> 86
<<interface>> 147, 365
<<manifest>> 395
<<merge>> 183
<<model>> 51, 178
<<represents>> 168
<<selection>> 266
<<source>> 390
<<topLevelPackage>> 98
<<use case>> 79
<<use>> 365
<<useCaseModel>> 98
графический 377
компонента 376
текстовый 376
Сторожевое условие 330
Структура:
внутренняя 154
композитная 155
Субъект 75
Сценарий 99

Т

Тело обработчика 280
Тип данных:
Boolean 47
Integer 48
String 48
UnlimitedNatural 49
Точка:
входа 324
выхода 324
расширения 88
Траектория 203
чередующаяся 203
Требование 97
Триггер 312

У

Узел 357, 384
выбора 319
глубокой истории 327
деятельности 247
завершения 319
защищенный 280
начальный 318
неглубокой истории 325
объекта 265
параметра деятельности 271
разделения 259, 322
решения 256
слияния 258
соединения 260, 321
среда выполнения 385
управления 254
устройство 387
финала деятельности 255
финала потока 256
Унифицированный язык моделирования (UML) 13
Устройство 387

Ф

Форма Бэкуса—Наура (БНФ) 54
Фрагмент взаимодействия 213
Функция, перегрузка 20

Х

Характеристика 106
поведения 106
структурная 106
Хранилище данных 268

Ц

Цель развертывания 388
Центральный буфер 267

Ч

Часть 156
Черный ящик 361

Ш

Шаблон 148
кооперации 170
Шлюз 234

Я

Язык объектных ограничений 52