Станислав Давыдов
Алексей Ефимов

bhv

IntelliJ IDEA

Профессиональное программирование на Java

- Описание интерфейса и настроек
- Эффективное создание и модификация кода
- Анализ кода на возможные ошибки
- Удобная работа с XML и компонентами J2EE
- Создание плагинов для IntelliJ IDEA

Наиболее полное руководство

+CD

В ПОДЛИННИКЕ®

Станислав Давыдов
Алексей Ефимов

IntelliJ IDEA

Профессиональное программирование на Java

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800java
ББК 32.973.26-018.2
Д13

Давыдов С. В., Ефимов А. А.

Д13 IntelliJ IDEA. Профессиональное программирование на Java. — СПб.: БХВ-Петербург, 2005. — 800 с.: ил.

ISBN 5-94157-607-2

Книга представляет собой исчерпывающее руководство программиста по использованию среды разработки IntelliJ IDEA для создания Java-приложений. Рассмотрены основные особенности работы с IntelliJ IDEA: создание кода на основе информации о классах, редактор с мощными вспомогательными возможностями, встроенная поддержка рефакторинга кода, средства структурного анализа и проверки кода на наличие логических ошибок, потенциально опасных конструкций и возможных способов улучшения, поддержка J2EE-приложений, поддержка XML, визуальный редактор пользовательского интерфейса, встроенная поддержка unit-тестирования с использованием JUnit, поддержка основных систем контроля версий (SourceSafe, CVS, Starteam), полная поддержка Java 5, поддержка плагинов. Особое внимание уделено практическим приемам работы с IntelliJ IDEA. Компакт-диск содержит все необходимое для разработки Java-приложений.

Для программистов

УДК 681.3.068+800java
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Алия Амирова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.04.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 64,5.
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 5-94157-607-2

© Давыдов С. В., Ефимов А. А., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Предисловие 15

От авторов 15
Для кого эта книга 16
Структура книги 16
Благодарности 18
Замечания и предложения 18

ЧАСТЬ I. ОБЩИЕ СВЕДЕНИЯ ОБ INTELLIJ IDEA 19

Глава 1. Прошлое и настоящее 21

1.1. Идеология работы с файлами 22
1.2. Логическая модель проекта 23
1.3. Идеология работы с пользователями 24

Глава 2. Установка и запуск 25

2.1. Регистрация 25
2.2. Комплект поставки 26
2.2.1. Установка под Windows 26
2.2.2. Установка под Linux 26
2.2.3. Установка под Mac OS X 27
2.2.4. Установка под Generic Unix 27
2.2.5. Структура директорий 27
2.3. Запуск 28
2.4. Директории настроек и временной информации 28
2.4.1. Директория с настройками 28
2.4.2. Директория с временными файлами 29

Глава 3. Интерфейс 30

3.1. Организация рабочего пространства приложения 30
3.2. Меню 32
3.2.1. Меню *File (Файл)* 32
3.2.2. Меню *Edit (Редактирование)* 42
3.2.3. Меню *Search (Поиск)* 50
3.2.4. Меню *View (Просмотр)* 62
3.2.5. Меню *Go To (Перейти к)* 71
3.2.6. Меню *Code (Код)* 75
3.2.7. Меню *Analyze (Анализ)* 86
3.2.8. Меню *Refactor (Рефакторинг)* 87
3.2.9. Меню *Build (Сборка)* 89
3.2.10. Меню *Run (Запуск)* 92
3.2.11. Меню *Tools (Инструменты)* 98
3.2.12. Меню *Window (Окно)* 108
3.2.13. Меню *Help (Помощь)* 110
3.3. Панель инструментов (*Tool Bar*) 115
3.4. Инструментальные окна (*Tool Window*) 116
3.4.1. Режимы отображения 117
3.4.2. Состояния окна 118
3.4.3. Статусы окна 118
3.4.4. Размещение кнопок вызова 118
3.4.5. Дополнительные команды управления 119
3.4.6. Настройки инструментальных окон 119
3.4.7. Поиск в инструментальных окнах 119
3.4.8. Базовые инструментальные окна 119
3.5. Панель статуса (Status Bar) 141
3.6. Окна редактирования 142
3.6.1. Закладки редакторов 142
3.6.2. Поля редактора 144
3.6.3. Дополнительные визуальные элементы 147
3.6.4. Команды редактора 147

Глава 4. Настройки 153

4.1. *IDE Settings (Настройки IDE)* 155
4.1.1. Диалог настроек *General (Общие)* 156
4.1.2. Диалог настроек *Appearance (Внешний вид)* 158
4.1.3. Диалог настроек *Editor (Редактор)* 161
4.1.4. Диалог настроек *Code Completion (Завершение кода)* 166
4.1.5. Диалог настроек *Errors (Ошибки)* 170
4.1.6. Диалог настроек *Colors & Fonts (Цвета и Шрифты)* 173
4.1.7. Диалог настроек *Global Code Style (Глобальные стили кода)* 179

4.1.8. Диалог настроек *JDK & Global Libraries (JDK и Глобальные библиотеки)* ... 195
4.1.9. Диалог настроек *Debugger (Отладчик)* ... 199
4.1.10. Диалог настройки *Resources (Ресурсы)* ... 202
4.1.11. Диалог настроек *File Types (Типы файлов)* ... 203
4.1.12. Диалог настроек *Local History (Локальная история)* ... 207
4.1.13. *Live Templates ("Живые шаблоны")* ... 208
4.1.14. Диалог настройки *File Templates (Шаблоны файлов)* ... 221
4.1.15. Диалог настройки *Keymap (Сочетания клавиш)* ... 228
4.1.16. Диалог настройки *External Tools (Внешние программы)* ... 234
4.1.17. Диалог настройки *Path Variables (Местоположения)* ... 241
4.1.18. Диалог *TODO (Заметки)* ... 242
4.1.19. Диалог настройки *Plugins (Плагины)* ... 247
4.1.20. Диалог настройки *Application Servers (Серверы приложений)* ... 250
4.1.21. Диалог настроек *Intention Power Pack (Дополнительный пакет "подсказок")* ... 255
4.2. Настройки проекта ... 258
4.2.1. Диалог настройки *Paths (Пути)* ... 258
4.2.2. Диалог настройки *Compiler (Компилятор)* ... 259
4.2.3. Диалог настройки *Version Control (Контроль версий)* ... 263
4.2.4. Диалог настройки *Project Code Style (Стиль кода проекта)* ... 264
4.2.5. GUI Designer ... 266

ЧАСТЬ II. СОЗДАНИЕ КОДА ... 267

Глава 5. Управление проектами ... 269

5.1. Структура проекта ... 269
5.2. Создание проекта ... 269
5.2.1. Выбор JDK ... 270
5.2.2. Выбор количества модулей ... 271
5.2.3. Выбор типа модуля ... 271
5.3. Редактирование свойств модуля ... 289
5.3.1. Закладка *Paths (Пути)* ... 290
5.3.2. Закладка *Libraries (Библиотеки)* ... 293
5.3.3. Закладка *Dependencies (Зависимости)* ... 296
5.3.4. Закладка *Order/Export (Порядок/Экспорт)* ... 296
5.3.5. Закладка *Javadoc* ... 298
5.3.6. Закладка *Web Module Settings (Свойства Web-модуля)* ... 299
5.3.7. Закладка *J2EE Build Settings (Настройка сборки J2EE)* ... 301
5.3.8. Закладка *EJB Module Settings (Свойства EJB-модуля)* ... 304
5.3.9. *J2EE Application Module Settings (Свойства модуля J2EE приложения)* ... 305
5.4. Создание проекта из имеющихся файлов ... 307

Глава 6. Работа с кодом 311

6.1. Работа с редактором 312
6.1.1. Завершение кода (completion) 329
6.1.2. Подсветка 334
6.1.3. "Подсказки" (intention actions) 336
6.1.4. Форматирование кода 341
6.1.5. Информация о коде 342
6.1.6. Буфер обмена 343
6.1.7. Сокращенное представление кода 343
6.2. Навигация по коду 344
6.2.1. Класс 344
6.2.2. Файл 344
6.2.3. Метод или поле 345
6.2.4. Переход к определению, имплементации, определению типа или базовому методу 345
6.2.5. Перемещение по коду "вперед-назад" 345
6.2.6. Список последних файлов 346
6.2.7. Закладки 346
6.2.8. Заметки (TODO) 346
6.2.9. Иерархия 346
6.3. Генерация кода 347
6.3.1. Шаблоны файлов 347
6.3.2. "Живые шаблоны" 347
6.3.3. Создание однотипных классов, интерфейсов и перечислений 347
6.3.4. Создание однотипных файлов 348
6.3.5. Создание методов *get* и *set* 348
6.3.6. Автоматическое создание конструкторов 348
6.3.7. Создание методов 348
6.3.8. Автоматическое создание блоков кода 348
6.3.9. Строки импорта 349
6.3.10. Комментарии 349
6.4. Анализ кода 350
6.4.1. Проверка кода (inspections) 350
6.4.2. Анализ зависимостей 386
6.4.3. Анализ дубликатов 390
6.5. Структурный поиск 392
6.5.1. Редактирование переменных 393
6.5.2. Доступные шаблоны 395
6.5.3. Результаты поиска 396
6.5.4. Структурная замена 396

Глава 7. Создание интерфейса пользователя 398

7.1. Формы 398
7.1.1. Создание формы 399

7.1.2. Свойства компонент 415
7.1.3. Размещение компонент 420
7.2. Создание кода 425
7.3. Добавление собственных компонент 427
7.4. Дополнительные возможности 430
7.4.1. Настройка общих свойств визуального редактора 430
7.4.2. *Data Binding Wizard (Мастер связей)* 433
7.4.3. Компиляция форм 436
7.4.4. Интернализация (i18n) 437

Глава 8. Рефакторинг 442

8.1. Рефакторинг *Renate (Переименовать)* 443
8.1.1. Переименование директорий 443
8.1.2. Переименование файлов 444
8.1.3. Переименование пакетов 444
8.1.4. Переименование классов 446
8.1.5. Переименование методов 446
8.1.6. Переименование полей класса 447
8.1.7. Переименование параметров вызова метода 448
8.1.8. Переименование локальных переменных 449
8.2. Рефакторинг *Change Signature (Изменить сигнатуру)* 449
8.3. Рефакторинг *Make Method Static (Сделать метод статичным)* 454
8.3.1. Простой случай 454
8.3.2. Сложный случай 456
8.4. Рефакторинг *Convert To Instance Method (Конвертировать в обычный метод)* 457
8.5. Рефакторинг *Move (Переместить)* 458
8.5.1. Перемещение директорий 459
8.5.2. Перемещение файлов 459
8.5.3. Перемещение пакетов 460
8.5.4. Перемещение классов 462
8.5.5. Перемещение членов класса 465
8.6. Рефакторинг *Copy (Копировать)* 467
8.6.1. Копирование директорий 467
8.6.2. Копирование файлов 468
8.6.3. Копирование классов 468
8.7. Рефакторинг *Safe Delete (Безопасное удаление)* 469
8.7.1. Удаление класса 469
8.7.2. Удаление метода 470
8.7.3. Удаление поля класса 470
8.8. Рефакторинг *Extract Method (Выделить метод)* 471
8.9. Рефакторинг *Replace Method Code Duplicates (Заменить методом повторяющийся код)* 474

8.10. Рефакторинг *Introduce Variable (Ввести переменную)* 476
8.11. Рефакторинг *Introduce Field (Ввести поле класса)* 477
8.11.1. Локальная переменная статического метода 477
8.11.2. Локальная переменная обычного метода 479
8.12. Рефакторинг *Introduce Constant (Ввести константу)* 480
8.13. Рефакторинг *Introduce Parameter (Ввести параметр)* 481
8.14. Рефакторинг *Extract Interface (Выделить интерфейс)* 482
8.15. Рефакторинг *Extract Superclass (Выделить базовый класс)* 485
8.16. Рефакторинг *Use Interface Where Possible (Использовать интерфейс, где это возможно)* 488
8.17. Рефакторинг *Pull Members Up (Перемещение членов класса вверх)* 490
8.18. Рефакторинг *Push Members Down (Перемещение членов класса вниз)* 492
8.19. Рефакторинг *Replace Inheritance With Delegation (Заменить наследование делегированием)* 495
8.20. Рефакторинг *Inline (Подстановка в код)* 497
8.20.1. Подстановка для метода 497
8.20.2. Подстановка для статического поля класса 499
8.21. Рефакторинг *Convert Anonymous to Inner (Конвертировать анонимный класс во внутренний)* 500
8.22. Рефакторинг *Encapsulate Fields (Инкапсулировать поля)* 502
8.23. Рефакторинг *Replace Temp with Query (Заменить временное использование вызовом)* 505
8.24. Рефакторинг *Replace Constructor With Factory Method (Заменить конструктор методом фабрики)* 507
8.25. Рефакторинг *Generify (Генерификация)* 508

Глава 9. Компиляция, запуск, отладка 512

9.1. Компиляция 512
9.1.1. Компиляция Java-классов 512
9.1.2. Компиляция RMI 514
9.1.3. Валидация JSP 514
9.2. Типы приложений 514
9.2.1. Java-приложения (закладка *Application*) 516
9.2.2. Апплеты (закладка *Applet*) 517
9.2.3. Тесты (закладка *JUnit*) 520
9.2.4. Удаленные приложения (закладка *Remote*) 524
9.2.5. Web-приложения для Tomcat (закладка *Tomcat Server*) 526
9.2.6. Приложения для JSR45-совместимых серверов (JSR-45 Compatible Server) 530
9.2.7. Приложения для сервера WebLogic (закладка *WebLogic Instance*) 533
9.3. Запуск 538
9.3.1. Java-приложения (закладка *Application*) 538
9.3.2. Апплеты 540

9.3.3. Тесты JUnit 544
9.3.4. J2EE-приложения 546
9.4. Отладка 549
9.4.1. Точки останова 550
9.4.2. Пошаговая отладка 561
9.4.3. Информация о потоках (закладка *Threads*) 569
9.4.4. Информация о переменных (закладка *Frame*) 571
9.4.5. *Watches (Наблюдения)* 576
9.4.6. Вычисление выражений 577
9.4.7. Функция HotSwap 579

Глава 10. J2EE 580

10.1. Работа с XML 580
10.1.1. Редактирование XML 580
10.1.2. Поддержка типизации 582
10.2. Серверы приложений 583
10.3. Работа с EJB 583
10.3.1. Создание EJB 585
10.3.2. Установка связей между EJB 598
10.4. Web-приложения 600
10.4.1. Сервлеты 603
10.4.2. Фильтры 605
10.4.3. JSP 606
10.5. Приложения J2EE 608

Глава 11. Поддержка новых возможностей Java 5 610

11.1. Generics 611
11.2. Цикл *for* для коллекций 612
11.3. Auto(un)boxing 613
11.4. Перечислимый тип *enum* 614
11.5. Переменное число параметров метода 616
11.6. Статические импорты 617
11.7. Аннотации 618

ЧАСТЬ III. ИНТЕГРАЦИЯ 621

Глава 12. Системы контроля версий 623

12.1. Утилита Diff 623
12.1.1. Сравнение двух файлов 623
12.1.2. Сравнение частей файлов 624
12.1.3. Представление результатов сравнения 624

12.1.4. Дополнительные команды 625
12.2. Локальная система контроля версий 625
12.2.1. Версии файла 626
12.2.2. Версии директорий 629
12.2.3. Метки 630
12.3. CVS 631
12.3.1. Настройки 631
12.3.2. Работа с проектом 641
12.3.3. Визуальное выделение файлов 648
12.3.4. Команды 648
12.4. SourceSafe 665
12.4.1. Настройки 665
12.4.2. Визуальное выделение файлов 667
12.4.3. Команды 668
12.5. StarTeam 672
12.5.1. Настройки 672
12.5.2. Визуальное выделение файлов 672
12.5.3. Команды 672
12.6. Дополнительные команды 676

Глава 13. Утилита Ant 677

13.1. Создание и подключение файлов сборки 677
13.2. Редактирование файлов сборки 680
13.2.1. Список доступных команд и параметров 680
13.2.2. Вывод дополнительной информации 681
13.2.3. Проверка ошибок 682
13.2.4. Редактирование зависимостей 682
13.2.5. Использование свойств 683
13.3. Настройка параметров 683
13.3.1. Общие свойства 684
13.3.2. Параметры для передачи файлу сборки 684
13.3.3. Выполнение 684
13.3.4. Дополнительные библиотеки 685
13.3.5. Фильтры 686
13.4. Привязка целей Ant 687
13.5. Запуск целей Ant 689
13.6. Генерация файла сборки для проекта 690

Глава 14. Тестирование 692

14.1. Представление тестов в проекте 692
14.2. Запуск тестов 693
14.3. Просмотр результатов 696

Глава 15. Плагины 698

15.1. Общие сведения о плагинах 698
15.1.1. Сообщество разработчиков плагинов IDEA 698
15.2. Разработка плагинов 698
15.2.1. Подготовка IntelliJ IDEA к разработке плагинов 699
15.2.2. Общая компонентная модель 709
15.2.3. Диагностика ошибок 715
15.2.4. Сохранение и восстановление состояния компонент 718
15.2.5. Система действий в IntelliJ IDEA 723
15.2.6. Стандартные визуальные компоненты IntelliJ IDEA 731
15.2.7. Редактор IntelliJ IDEA 746
15.2.8. Виртуальная файловая система IntelliJ IDEA (VFS) 763
15.2.9. Система контроля версий IntelliJ IDEA (VCS) 766
15.2.10. Структурный программный интерфейс IntelliJ IDEA (PSI) 768
15.3. Структура директорий плагина 771
15.4. Дескриптор плагина 772
15.4.1. Требования к дескрипторам 772
15.4.2. Общая информация о плагине 772
15.4.3. Описания компонент 774
15.4.4. Описание действий 774
15.5. Заключение 777

ЧАСТЬ IV. БЛИЖАЙШЕЕ БУДУЩЕЕ 779

Глава 16. IDEA 5 781

16.1. Поддержка XML, HTML, CSS 781
16.1.1. XML 781
16.1.2. HTML 782
16.1.3. CSS 782
16.2. Поддержка J2EE 782
16.3. Работа с проектом 783
16.4. Настройки 783
16.5. Рефакторинги 783
16.6. Системы контроля версий 784
16.7. Плагины и OpenAPI 784
16.8. Поддержка J2ME 785
16.9. Отладка 785
16.10. Дополнительные функции редактора 785
16.11. Прочие улучшения 785

Заключение 787

Список ссылок 789

Содержимое компакт-диска 791

Предметный указатель 792

В 1900 году лорд Кельвин, выступая в Лондонском Королевском Обществе в канун Нового года, сказал: "Я удовлетворен современным состоянием науки. Все основные законы открыты, и научный небосвод чист. За исключением двух небольших облачков".

В 2000 году на рынок постепенно выходили Java IDE (Integrated Development Environment — интегрированные среды разработки) больших уважаемых гигантов: Borland, Symantec, Sun. Разработкой IDE заниматься было не модно, все равно, что начинать писать свой Windows — какой смысл, если все уже, в общем-то, сделано и открыто до нас.

Как раз тогда возникла Intellij IDEA и за полтора года, по общему признанию, опередила по качеству неповоротливых монстров. Это теперь мы со смехом вспоминаем убогость тогдашних "систем четвертого поколения" и уже точно знаем, что и теперешним их потомкам есть куда развиваться. А словосочетание "интегрированная среда разработки" сейчас чем-то напоминает "ЭВМ": перед глазами встает машинный зал с рядами крутящихся бобин с интеллектом нынешнего калькулятора.

Чем Intellij IDEA выделяется среди аналогичных IDE? Прежде всего, как неоднократно замечают пользователи, тем, что она "does not get in the way", то есть не диктует свой способ, свой стиль использования, а, наоборот, помогает программисту решать свои задачи с наименьшими усилиями. Ну и, наконец, спросите у фанатов Apple Computer, почему им приятно брать в руки свои Маки?

— Ну, фиг его знает. Это так стильно!

Я думаю, Стас Давыдов сумел передать в своей книге это ощущение стиля, другой философии работы с Intellij IDEA.

Алексей Кудрявцев, Senior Software Engineer, JetBrains, Inc.

Предисловие

От авторов

С момента появления IntelliJ IDEA на свет в 2000 году прошло уже достаточно много времени. За это время IntelliJ IDEA превратилась в серьезный продукт, обеспечивший Java-разработчиков практически всем необходимым для комфортной работы. Однако до сих пор об IntelliJ IDEA не было написано ни одной книги. Мы хотим исправить эту, с нашей точки зрения, ошибку.

Одна из причин, побудивших нас написать эту книгу — это то, что IntelliJ IDEA обладает колоссальными возможностями, о которых мало кто знает, даже многие разработчики из самой компании JetBrains. Но мы даже не предполагали, сколько на самом деле этих возможностей, скрытых в глубинах функциональности IntelliJ IDEA!

Книга, которую вы держите в руках, — первая книга в мире, целиком и полностью посвященная IntelliJ IDEA.

IntelliJ IDEA имеет англоязычный интерфейс, поэтому при описании элементов интерфейса, названий и понятий нам пришлось переводить их на русский язык самостоятельно. Мы старались подбирать как можно более точный и понятный перевод терминов, однако в нескольких случаях нам это сделать не удалось. Так, к примеру, остались без перевода английские термины generics и auto(un)boxing. Также мы полностью оставили без перевода термины J2EE, поскольку, как мудро заметил один из наших технических помощников в этой области, в таком переводе можно дойти до абсурда (в качестве примера он привел совершенно дурной перевод термина "Message Driven Bean" — "Боб, погоняемый сообщением", который он видел в одной из книг по J2EE).

Для кого эта книга

Прежде чем начать эту книгу, мы долго думали, для кого мы будем ее писать.

С одной стороны, хотелось рассказать обо всех тонкостях работы с IntelliJ IDEA как можно подробнее, чтобы даже не очень опытные программисты (или незнакомые с IntelliJ IDEA) смогли легко использовать ее в своей повседневной работе и получать от этого максимум удовольствия и результата. Однако в этом случае книга получилась бы очень большой и, возможно, не очень интересной для других программистов — тех, кто либо уже работают с IntelliJ IDEA, либо имеют богатый опыт работы с другими средствами разработки. К тому же у нас ушло бы на ее написание слишком много времени — информация могла бы уже устареть.

С другой стороны, ограничив материал книги лишь разными "продвинутыми" техниками работы с IntelliJ IDEA, мы бы потеряли наших возможных читателей из числа начинающих Java-разработчиков.

Поэтому мы решили пойти на компромисс с собой и сделать следующее: разбить книгу на три части, в первой из которых информация дается максимально подробно, с примерами и пояснениями. Она будет понятна даже начинающим разработчикам. Вторая часть несколько более сложная и требует от читателя знаний в области Java и смежных технологиях, таких, например, как рефакторинг или J2EE. Третья часть носит справочный характер — каждый может взять из нее то, что ему нужно — информацию об интеграции с системами контроля версий, утилитами сборки и тестирования. В последней главе третьей части содержится информация о системе плагинов IntelliJ IDEA. Эта глава будет интересна Java-разработчикам, создающим модули расширения для IntelliJ IDEA.

Структура книги

Информация об IntelliJ IDEA структурирована в виде шестнадцати глав, разделенных на четыре части.

Первая часть включает главы с первой по четвертую.

Глава 1 содержит общую информацию об IntelliJ IDEA, так сказать для общего развития ☺.

Глава 2 повествует об установке IntelliJ IDEA, а также о различных особенностях, сопровождающих этот процесс на различных операционных системах.

Глава 3 содержит подробное описание интерфейса IntelliJ IDEA, включая описание системы меню, панели инструментов, команд, инструментальных окон и окон редактора. Информация в данной главе организована таким образом, чтобы ее можно было легко использовать в качестве справочного материала при работе с IntelliJ IDEA.

Глава 4 содержит подробные описания всевозможных настроек IntelliJ IDEA (коих не так уж и мало). Данную главу также можно использовать в качестве справочника.

Вторая часть содержит главы с пятой по одиннадцатую.

Глава 5 рассказывает об управлении программными проектами в рамках IntelliJ IDEA. Из этой главы вы узнаете о структуре проекта IntelliJ IDEA и его составных частях — модулях.

Глава 6 содержит информацию, полезную с практической стороны: здесь описана работа с редактором (включая основные моменты работы с редактором при написании кода), навигация, генерация и анализ кода. В этой главе также рассказано о функции структурного поиска и замены.

Глава 7 повествует о встроенном визуальном редакторе IntelliJ IDEA — средстве для создания интерфейса пользователя с помощью технологии Swing.

Глава 8 содержит справочную информацию по функциям IntelliJ IDEA, связанным с рефакторингом кода.

Глава 9 рассказывает о компиляции и сборке проектов, а также о запуске и отладке различных типов Java-приложений (в том числе J2EE-приложений).

Глава 10 содержит справочную информацию о поддержке J2EE в IntelliJ IDEA. В главе рассмотрены вопросы создания и настройки компонентов EJB, Web- и J2EE-приложений.

Глава 11 повествует о поддержке новых возможностей языка Java версии 5 в IntelliJ IDEA, таких как параметризированные классы (`generics`), классы-перечисления (`enum`) и т. д.

Третья часть книги содержит главы с двенадцатой по пятнадцатую.

Глава 12 рассказывает о взаимодействии IntelliJ IDEA с системами контроля версий: встроенной системе, CVS, SourceSafe и StarTeam.

Глава 13 содержит информацию о поддержке утилиты сборки Java проектов Ant.

Глава 14 повествует о поддержке в IntelliJ IDEA unit-тестирования Java-кода с помощью утилиты JUnit.

Глава 15 содержит информацию о системе плагинов IntelliJ IDEA — специальных подключаемых модулях расширения функциональности IntelliJ IDEA.

Последняя, четвертая часть содержит *главу 16*, рассказывающую о новых возможностях, которые будут доступны в следующей (пятой) версии IntelliJ IDEA, которая должна выйти в начале лета 2005 года.

В конце книги приведен алфавитный указатель.

Благодарности

Мы хотим выразить нашу признательность людям и компаниям, благодаря которым эта книга смогла увидеть свет.

Во-первых, мы бы хотели поблагодарить компанию JetBrains, Inc. и лично Евгения Беляева, Сергея Дмитриева и Валентина Кипяткова — создателей этого замечательного программного продукта IntelliJ IDEA.

Во-вторых, мы хотим поблагодарить издательство "БХВ-Петербург" за то, что оно поверило в наш проект и претворило его в жизнь, издав эту книгу, а также лично наших редакторов: Игоря Шишигина и Алию Амирову.

В-третьих, мы хотим поблагодарить всех людей, которые помогали нам при написании этой книги: Сергея Баранова и Алексея Кудрявцева из компании JetBrains за ценные советы и ответы на наши вопросы, возникавшие по ходу работы над книгой; Антона Катилина из компании YourKit LLC за ценные замечания по поводу описания визуального редактора IntelliJ IDEA; активных читателей "Живого Журнала", посвященного созданию этой книги, за ценные советы: insolite, nettoyeur и др.

Кроме этих людей, помогавших нам в создании этой книги, у каждого из нас найдется, кого поблагодарить лично.

Станислав Давыдов: мне хотелось бы отдельно поблагодарить Гоблина и Гнума, которые стойко терпели мой стук по клавишам в течение долгой работы над книгой.

Алексей Ефимов: я бы хотел поблагодарить Юльку за терпение и понимание, Дмитрия Кашина, создателя плагина Properties Editor, который помогал в разрешении некоторых вопросов и давал ценные замечания, и Гоблинов, без которых вообще ничего не получилось бы.

Извините, если мы кого-то забыли упомянуть здесь. Сообщите нам об этом, чтобы в следующем издании мы исправили эту ошибку.

Станислав Давыдов (**davidovsv@yandex.ru**)

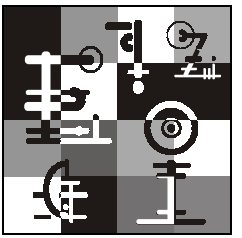
Алексей Ефимов (**aefimov-box@yandex.ru**)

Санкт-Петербург — Москва, 2005 год.

Замечания и предложения

Если у вас возникнут любые замечания или предложения по поводу материалов, представленных в этой книге, мы будем рады получить от вас эту информацию. Также мы будем признательны за любые дополнения или информацию об ошибках и неточностях.

Вы можете обращаться к нам лично по электронной почте, либо использовать сообщество в "Живом Журнале", посвященное этой книге (**http://www.livejournal.com/community/ideabook**).



Часть I

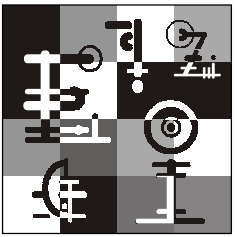
Общие сведения об IntelliJ IDEA

Глава 1. Прошлое и настоящее

Глава 2. Установка и запуск

Глава 3. Интерфейс

Глава 4. Настройки



Глава 1

Прошлое и настоящее

IntelliJ IDEA — очень молодое и динамично развивающееся Java IDE (Integrated Development Environment — интегрированная среда разработки). Молодость в данном случае выступает в роли достоинства — программа еще не успела обрасти пережитками, тянущимися со старых версий. Динамичность же обеспечивается новейшими методиками разработки, которые использует команда компании JetBrains, Inc., разрабатывающая IntelliJ IDEA.

По сравнению с известными мировыми брендами в этой области с точки зрения рядовых Java-разработчиков IntelliJ IDEA выглядит чрезвычайно привлекательно, поскольку старается воплотить все новые веяния в процессе создания программного обеспечения на Java. Кроме того, она, ориентируясь больше на ручное написание (в отличие от визуальных систем), позволяет писать код максимально быстро, удобно и без ошибок. Даже если от IntelliJ IDEA оставить один лишь редактор, она все равно останется чрезвычайно полезной.

По заявлению компании, *IntelliJ IDEA создается Java-разработчиками для Java-разработчиков.* Стоит лишь несколько часов пописать код в IntelliJ IDEA, как становится понятно, что это действительно так. Если продолжить писать код в IntelliJ IDEA около недели, окажется, что все остальные среды разработки выглядят громоздкими и неудобными. Авторы сами убедились в этом в свое время.

Кроме богатых возможностей редактирования кода, в IDEA есть много других возможностей. Из числа наиболее полезных вещей можно назвать встроенную поддержку рефакторинга, гибкие возможности по запуску и отладке приложений, включая возможности по загрузки измененного кода "на лету" — HotSwap, отладку J2EE-приложений и многое другое.

Долгое время в IDEA не было вообще никаких визуальных средств, однако, начиная с четвертой версии появился встроенный визуальный редактор форм (использующий Swing).

1.1. Идеология работы с файлами

В IntelliJ IDEA используется очень дружественная к пользователю система работы с файлами. Если вы обратите внимание на меню IntelliJ IDEA, там не обнаружится многочисленных команд сохранения файлов. Одна-единственная команда **Save All** (Сохранить все) (*см. раздел 3.2.1.13*) скорее дань психологическому спокойствию разработчика, нежели реальная необходимость.

Отсутствие команд сохранения неслучайно.

За историю существования программ, предназначенных для изменения каких-либо данных, разработчики этих программ приучили нас сохраняться каждые несколько минут. Иначе, малейший сбой в программе или компьютере — и результат целого дня работы может быть потерян. Данный подход к организации сохранения изменений, произошедших в результате работы программы, чрезвычайно неудобен.

Другой подход — сохранять все и всегда — впервые в массовом масштабе был реализован в портативных устройствах PDA (Personal Data Assistant — персональный помощник) или КПК (карманный персональный компьютер). Суть его весьма проста и исходит из архитектуры КПК, в которых память, используемая для размещения программ и данных, также является и оперативной памятью. То есть если на КПК некоторый файл загружен в редактор и там изменяется, то он также изменяется и в месте своего постоянного хранения. Это достаточно грубое описание, однако для обычного пользователя все выглядит именно так.

При таком подходе пользователю просто не нужно беспокоиться, сохранил он файл или нет.

В IntelliJ IDEA использован именно такой подход — избавление программиста от лишних действий по сохранению файлов, т. к. IntelliJ IDEA все файлы сохраняет сама. Делать она это может по трем событиям:

- потеря фокуса окна IntelliJ IDEA;
- периодически, в случае, если приложение неактивно (отсутствуют события с клавиатуры или мыши);
- по команде пользователя **File | Save All** (Файл | Сохранить все) или при нажатии клавиш <Ctrl>+<S>.

А чтобы не беспокоиться, не затерли ли мы какой-нибудь важный файл, IDEA ведет журнал всех изменений. Эта система называется LVCS (Local version control system или просто Local History) (*см. раздел 12.2*). Она позволяет просматривать изменения в файлах, возвращать к жизни давно забытые идеи и легко отвечать на вопросы типа: "Покажи все изменения в моем классе, произошедшие со времени последнего успешного запуска тестов".

1.2. Логическая модель проекта

Основой модели разработки программ в IntelliJ IDEA является *проект* (Project). Проект — это некоторый набор файлов (классов, ресурсов, JSP-страниц и пр.) и настроек, актуальных для всех этих файлов. Все файлы в проекте объединены в *модули*. Модуль — это структурная единица проекта, которая позволяет разделять файлы проекта некоторым логическим образом, а также определить для них некоторые специфичные настройки, отличные от общих настроек проекта.

Проще всего назначение модулей можно понять из следующего примера. Допустим, мы пишем некоторое Web-приложение, которое состоит из JSP-файлов, а также нескольких сервлетов, предназначенных для авторизации. Кроме самих классов с сервлетами, имеется много вспомогательных классов, необходимых для функционирования самих сервлетов. Будет логично, если в нашем проекте мы выделим два модуля:

1. Модуль с сервлетами и сопутствующими классами (в IntelliJ IDEA такой модуль называется Java Module).
2. Модуль с JSP-файлами (в IntelliJ IDEA такой модуль называется Web-модуль).

Почему было выбрано именно такое разделение? Во-первых, наши классы сервлетов работают с собственными библиотеками (т. е. должны компилироваться вместе с ними), во-вторых, мы хотим, чтобы после сборки приложения все классы были объединены в некоторую отдельную библиотеку (JAR-файл), которая просто размещается в специфическом для Web-приложения месте (папка /WEB-INF/lib). В-третьих, все остальные составляющие нашего Web-приложения — JSP-файлы, не имеют прямой связи с классами из этой библиотеки, однако работают совместно с ними.

Такое разделение — не единственно возможное. В принципе, можно было просто "свалить" все файлы в один модуль, однако тогда пострадала бы логическая модель представления нашего приложения. Чем проще и нагляднее представлен наш проект, тем меньше вероятность допустить в нем какие-то ошибки, связанные с его структурной организацией. При грамотном разделении проекта на модули IntelliJ IDEA сделает некоторую работу за вас, например, проверит целостность ссылок между различными классами проекта, а также позволит еще до этапа компиляции обнаружить некоторые ошибки (если они, конечно, есть ☺).

Количество модулей в проекте не ограничено. Чем сложнее приложение, тем больше в нем модулей. Например, в проекте самой IntelliJ IDEA на момент написания этой главы содержится порядка 50 модулей.

1.3. Идеология работы с пользователями

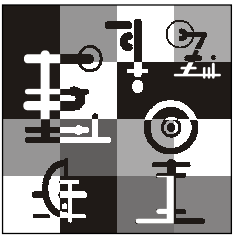
Каким бы ни был замечательным программный продукт, в нем всегда найдутся ошибки. Каждый, кто работает с любым программным продуктом, рано или поздно замечает, что одни вещи можно было бы сделать более удобными, другие — более быстрыми, третьи вообще неплохо было бы просто иметь в продукте.

IntelliJ IDEA — коммерческий программный продукт, созданный специально для Java-разработчиков. Основная цель его создания — сделать работу Java-программистов максимально удобной, качественной и быстрой. Компания JetBrains, Inc. — создатель IntelliJ IDEA — прикладывает к этому все силы.

Разработка самой IntelliJ IDEA построена по очень интересному принципу. Во-первых, каждый пользователь IntelliJ IDEA может попросить разработчиков включить в IntelliJ IDEA ту функциональность, которая ему нужна. Для таких запросов используется система онлайн учета запросов необходимых изменений (Change Request). Все пожелания пользователей учитываются, систематизируются и самые важные (те, о которых попросило максимальное количество пользователей) включаются в следующую версию IntelliJ IDEA. То есть фактическим заказчиком разработки программного продукта в данном случае может выступать каждый из нас — самый обычный Java-разработчик.

Все ошибки, которые возникают у разработчиков при работе IntelliJ IDEA также собираются и учитываются в специальной онлайн-системе (Bug Tracker). Ошибки оперативно исправляются и периодически выпускаются обновления той или иной версии продукта.

Благодаря тесной работе с пользователями IntelliJ IDEA, продукт получается удобным и качественным. Да, сама IntelliJ IDEA разрабатывается с помощью IntelliJ IDEA — это тоже немаловажный момент.



Глава 2

Установка и запуск

В данной главе будет рассказано о том, откуда можно скачать IntelliJ IDEA, как ее установить и запустить.

2.1. Регистрация

IntelliJ IDEA является коммерческим программным продуктом (стоимость однопользовательской коммерческой лицензии составляет порядка $500, а некоммерческой — для образовательных учреждений — $100). Тем не менее компания JetBrains, производящая IntelliJ IDEA, предоставляет возможность любому желающему бесплатно попробовать поработать в IntelliJ IDEA в течение 30 дней. Для этого нужно зарегистрироваться на сайте компании (**http://www.jetbrains.com**) и получить на указанный электронный адрес 30-дневный лицензионный ключ, который активирует пробную лицензию (evaluation license). Эта лицензия ограничивает лишь срок использования продукта, но никак не ограничивает функциональность.

Подробнее о регистрации лицензии IntelliJ IDEA можно прочитать в *разделе 3.2.13.5*.

Кроме платной версии IntelliJ IDEA, имеется еще и так называемая предварительная версия, которую можно получить, участвуя в "Программе предварительного доступа" (Early Access Program — EAP). Данная программа дает возможность любому желающему использовать версию IntelliJ IDEA, которая только разрабатывается.

Например, на момент написания данной книги в качестве такой предварительной версии для EAP предоставляется IntelliJ IDEA 5.0 ("Irida" code name).

EAP-версия может использоваться на всем протяжении разработки очередной версии IntelliJ IDEA (т. е. не менее полугода) без ограничений. Конечно, данная версия содержит значительно больше ошибок по сравнению со стабильной коммерческой версией. Тем не менее EAP-версия также

содержит и функции, запланированные на новую версию IntelliJ IDEA, и поэтому пользуется немалой популярностью.

EAP-версию IntelliJ IDEA можно скачать с сайта IntelliJ Technology Network (ITN — **http://www.intellij.net**), для чего необходимо предварительно зарегистрироваться на этом сайте. По сути, EAP делает каждого пользователя новой версии добровольным бета-тестером продукта в обмен на возможность его бесплатного использования.

Каждый участник EAP получает в свое распоряжение лицензию на использование IntelliJ IDEA сроком на один месяц, которая продлевается каждый месяц (т. е. пользователю необходимо заходить на сайт ITN минимум раз в месяц).

2.2. Комплект поставки

IntelliJ IDEA — приложение, целиком и полностью написанное на Java, соответственно, оно может работать на любой платформе, на которой имеется JRE (Java Runtime Environment). Минимальная версия JRE, которая требуется для работы IntelliJ IDEA — JRE 1.4.2_04. Для компиляции Java-классов также требуется наличие JDK (Java Development Kit).

Для операционных систем семейства Windows и Linux IntelliJ IDEA поставляется в виде самораспаковывающегося дистрибутива, содержащего программу-инсталлятор.

В комплекте к некоторым дистрибутивам, кроме основного приложения — IntelliJ IDEA, прилагается и необходимое Java окружение (JRE) для запуска IntelliJ IDEA.

2.2.1. Установка под Windows

Для операционной системы Windows дистрибутив IntelliJ IDEA занимает порядка 56 Мбайт. Дистрибутив представляет собой одиночный EXE-файл с инсталлятором и IntelliJ IDEA (например, idea-4.5.3.exe).

Процесс инсталляции вполне стандартен. Если на вашем компьютере установлены предыдущие версии IntelliJ IDEA, указав инсталлятору соответствующую опцию, можно импортировать настройки от старой версии.

В комплекте с дистрибутивом поставляется JRE версии 1.4.2_04 (для IntelliJ IDEA версии 4.5.3).

2.2.2. Установка под Linux

Для операционной системы Linux существует два вида дистрибутива IntelliJ IDEA:

- архив с директорий установки IntelliJ IDEA;
- дистрибутив, поставляемый в виде одиночного самораспаковывающегося файла (например, idea-4.5.3.bin).

2.2.2.1. Дистрибутив tar.gz

Первый вариант прост в установке — достаточно лишь выполнить команду разархивации `tar -xvzf idea-4.5.3.tar.gz` в требуемом каталоге.

В комплекте с данным дистрибутивом JRE не поставляется, поскольку этот дистрибутив также используется для установки IntelliJ IDEA на прочих Unix-совместимых платформах (*см. раздел 2.2.4*). Поэтому JRE необходимо установить отдельно.

JRE следует установить в директорию jre, расположенную внутри корневой директории с установленной IntelliJ IDEA.

Данный дистрибутив занимает около 40 Мбайт.

2.2.2.2. Самораспаковывающийся дистрибутив

Данный дистрибутив содержит кроме самой IntelliJ IDEA программу-инсталлятор, производящую все необходимые действия по установке, включая импорт настроек из старых версий IntelliJ IDEA и создание ссылок на программу запуска IntelliJ IDEA.

В составе данного дистрибутива имеется JRE версии 1.4.2_04.

Размер данного дистрибутива составляет около 59 Мбайт.

2.2.3. Установка под Mac OS X

Для операционной системы Mac OS X дистрибутив IntelliJ IDEA представляет собой файл типа Disk Image (dmg), который может быть установлен в систему стандартным образом.

Для установки под Mac OS X необходимо дважды щелкнуть левой кнопкой мыши на дистрибутиве, после чего выполнится стандартная операция Disk Copy, которая смонтирует диск с IntelliJ Labs.

Размер дистрибутива составляет порядка 34 Мбайт.

2.2.4. Установка под Generic Unix

Если некоторая операционная система Unix имеет поддержку Java и JDK версии не ниже, чем 1.4.2_04, на данную систему можно установить IntelliJ IDEA из дистрибутива tar.gz, как было описано в *разделе 2.2.2.1*.

Нам доподлинно известно, что IntelliJ IDEA успешно запускали на операционных системах Sun Solaris и FreeBSD.

2.2.5. Структура директорий

После установки IntelliJ IDEA создается следующая структура директорий:

- bin/ — файлы, необходимые для запуска IntelliJ IDEA;

- help/ — файлы справочной системы;
- jre/ — JRE;
- lib/ — файлы библиотек, необходимых для работы IntelliJ IDEA;
- license/ — файлы с лицензиями IntelliJ IDEA и библиотек;
- plugins/ — модули-расширения IntelliJ IDEA;
- redist/ — библиотеки, необходимые для сборки визуальных форм IntelliJ IDEA сторонними средствами;
- build.txt — информация о порядковом номере сборки IntelliJ IDEA;
- changes.txt — информация о внесенных изменениях;
- features.txt — информация о возможностях IntelliJ IDEA;
- knownissues.txt — информация об известных ошибках и проблемах;
- readme.txt — краткая информация.

2.3. Запуск

В зависимости от операционной системы запуск IntelliJ IDEA производится:

- на Windows с помощью файла idea.exe или idea.bat, расположенного в каталоге bin;
- на Linux и прочих Unix-системах с помощью файла idea.sh в каталоге bin;
- на Mac OS X — после того как образ диска смонтирован в системе (*см. раздел 2.2.3*), IntelliJ IDEA можно запускать прямо с этого смонтированного диска.

2.4. Директории настроек и временной информации

IntelliJ IDEA имеет две дополнительные директории, которые обычно создаются в пользовательской директории:

- директория с индивидуальными пользовательскими настройками IntelliJ IDEA — USER_HOME/.IntelliJIDEA/config;
- директория с временными файлами, используемыми для работы IntelliJ IDEA — USER_HOME/.IntelliJIDEA/system.

2.4.1. Директория с настройками

Пользовательские настройки хранятся в формате XML и, в принципе, могут быть модифицированы в любом текстовом редакторе, однако мы бы не советовали этого делать — лучше доверить это IntelliJ IDEA. Также в директории

настроек находится файл с лицензионным ключом к IntelliJ IDEA. Для IntelliJ IDEA версий 4.0 и выше этот файл называется idea40.key.

Кроме настроек, в поддиректории config/plugins хранятся плагины IntelliJ IDEA, установленные пользователем.

2.4.2. Директория с временными файлами

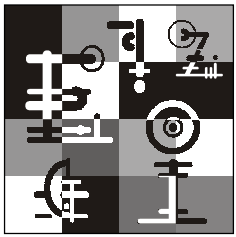
В директории с временными файлами содержатся различные кэшированные данные, которыми приложение IntelliJ IDEA управляет самостоятельно. Объем, занимаемый этими файлами, может достигать весьма внушительных размеров. Так, например, при работе над проектом с примерами к этой книге (достаточно небольшой проект) временные файлы заняли порядка 256 Мбайт. При работе над большими проектами временные файлы могут занимать значительно больше памяти.

Кроме кэшированных данных среди временных файлов содержится также репозиторий локальной системы контроля версий, встроенной в IntelliJ IDEA (*см. раздел 12.2*).

Мы бы не советовали удалять временные файлы без лишней необходимости. Во-первых, они существенно ускоряют работу IntelliJ IDEA, а во-вторых, даже если эти файлы удалить, IntelliJ IDEA создаст их вновь после следующего старта и открытия проекта.

Среди временных файлов в директории system/log хранятся также журнальные файлы работы IntelliJ IDEA, которые могут быть использованы для диагностирования неисправностей и ошибок в работе программы.

Обычно, в случае серьезной ошибки служба технической поддержки IntelliJ IDEA просит прислать эти файлы.



Глава 3

Интерфейс

Пользовательский интерфейс IntelliJ IDEA представляет собой стандартную форму MDI[1]-интерфейса, в котором общее окно соответствует одному проекту.

3.1. Организация рабочего пространства приложения

Общий вид окна проекта в IntelliJ IDEA представлен на рис. 3.1.

Кроме стандартного для большинства приложений меню, панели инструментов (ToolBar), панели статуса (StatusBar) и рабочей области, в IntelliJ IDEA есть интересное интерфейсное дополнение — *инструментальные окна* (ToolWindow — по аналогии с панелью инструментов — ToolBar). Инструментальные окна имеют кнопки вызова, расположенные по периметру рабочей области, с пиктограммой, подписью и числовым обозначением (последнее не обязательно). Если на эти кнопки нажать, рядом с ними откроются окошки с некоторой вспомогательной функциональностью (рис. 3.2). Подробнее об инструментальных окнах написано в *разделе 3.4*.

Обзор интерфейса IntelliJ IDEA мы начнем с меню, как с наиболее часто употребляемого списка команд, доступных в IntelliJ IDEA.

[1] MDI — Multi Document Interface — многодокументный интерфейс.

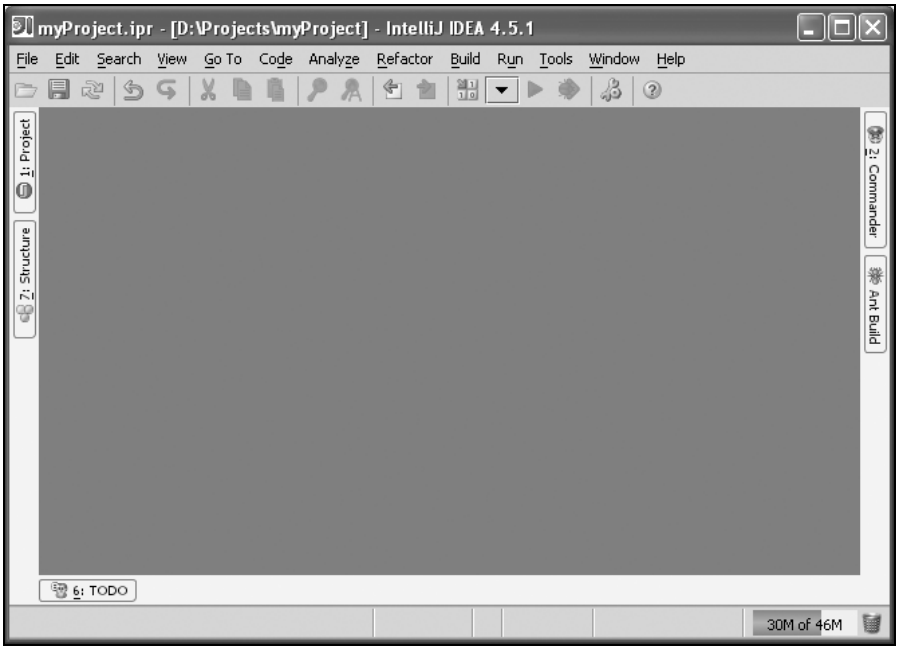


Рис. 3.1. Общий вид окна проекта в IntelliJ IDEA

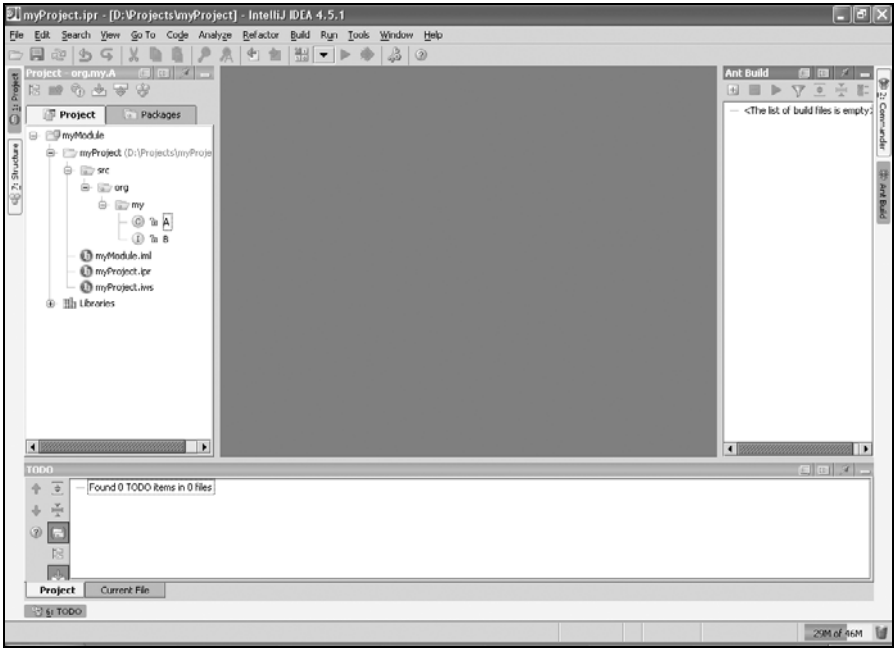


Рис. 3.2. Инструментальные окна

3.2. Меню

Главное меню IntelliJ IDEA состоит из следующих пунктов:

- **File** (Файл) — команды для работы с проектами, модулями, файлами;
- **Edit** (Редактирование) — стандартные команды для редактирования;
- **Search** (Поиск) — различные типы поиска и замены;
- **View** (Просмотр) — команды для отображения различных элементов интерфейса;
- **Go To** (Перейти) — команды навигации по проекту;
- **Code** (Код) — вспомогательные команды для написания кода;
- **Analyze** (Анализ) — команды для анализа кода;
- **Refactor** (Рефакторинг) — набор команд для проведения рефакторинга;
- **Build** (Сборка) — команды для компиляции и сборки проекта;
- **Run** (Запуск) — запуск и отладка приложений;
- **Tools** (Утилиты) — набор различных вспомогательных утилит;
- **Window** (Окно) — работа с окнами в IntelliJ IDEA;
- **Help** (Помощь) — контекстная помощь, справка о программе.

Пункты меню могут быть связаны с кнопками на панели инструментов, а также иметь *"горячие" клавиши* (hot keys) для быстрого вызова команд. В IntelliJ IDEA любой команде в меню можно назначить "горячую" клавишу. Для некоторых наиболее часто используемых команд "горячие" клавиши заданы по умолчанию. В описании таких команд мы будем указывать соответствующие им "горячие" клавиши в тексте сразу после названия.

Далее мы рассмотрим все приведенные пункты меню. Некоторые команды, присутствующие в меню, имеют весьма сложную функциональность, например, команды структурного поиска или рефакторинга. Таким командам посвящены отдельные главы или разделы, а в этой главе мы лишь приведем их краткое описание.

3.2.1. Меню *File (Файл)*

В меню **File** (Файл) представлены команды, с помощью которых можно выполнить некоторые действия с файлами — открыть, закрыть, вывести на печать и пр.

3.2.1.1. *New Project (Новый проект)*

Команда создания нового проекта (запускается мастер создания нового проекта, *см. раздел 5.2*).

3.2.1.2. *New Module (Новый модуль)*

Команда создания нового модуля в текущем проекте (запускается мастер создания нового модуля, *см. раздел 5.3*). Если ни одного проекта не открыто, эта команда недоступна.

3.2.1.3. *Open Project (Открыть проект)*

Команда открытия нового проекта из сохраненного на диске файла (рис. 3.3). Если один (или более) проект уже открыт, IntelliJ IDEA спросит, открывать ли новый проект в новом окне, либо в уже открытом.

3.2.1.4. *Open File (Открыть файл)*

Команда открытия файла в редакторе IntelliJ IDEA. Если открывается проектный файл, IntelliJ IDEA предложит открыть его как проект, либо как обычный файл.

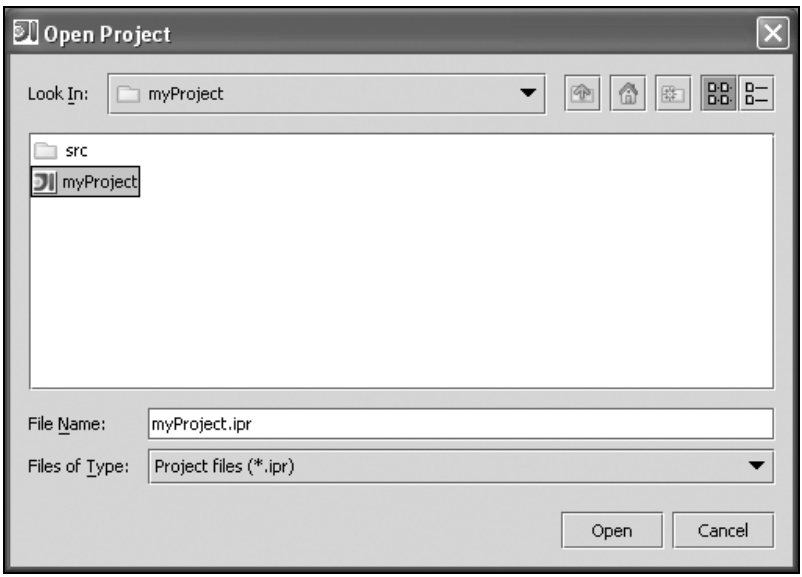


Рис. 3.3. Открытие проекта из сохраненного файла

Файлы проектов IntelliJ IDEA имеют расширение ipr. Если попытаться открыть такой файл с помощью команды **Open File** (Открыть файл), IntelliJ IDEA задаст вопрос о том, должен ли такой файл быть открытым как проект — полный аналог команды **Open Project** (Открыть проект), либо как обычный текстовый файл.

Надо заметить, что IntelliJ IDEA может открывать только те файлы, которые она "знает", т. е. те файлы, расширения которых описаны в типах файлов

в настройке **File Types** (Типы файлов) среди общих настроек IntelliJ IDEA (*см. раздел 4.1.11*).

3.2.1.5. *Reopen (Открыть заново)*

Подраздел меню **File** (Файл) содержит список проектов, с которыми вы недавно работали. IntelliJ IDEA запоминает последние 15 открытых проектов.

3.2.1.6. *Close Project (Закрыть проект)*

Команда закрывает открытый проект. Если ни одного открытого проекта нет, команда недоступна.

3.2.1.7. *Close Active Editor (Закрыть активную закладку редактора)*

"Горячие" клавиши: <Ctrl>+<F4>.

Команда закрывает закладку редактора, активную в данный момент (*см. раздел 3.6*). Этой команде соответствуют "горячие" клавиши: <Ctrl>+<F4>.

3.2.1.8. *Close All Editors (Закрыть все окна редактора)*

Команда закрывает все открытые окна редактора.

3.2.1.9. *Close All Editors But Active (Закрыть все окна редактора за исключением активного)*

Команда закрывает все открытые окна редактора за исключением активного.

3.2.1.10. Команды разделения и объединения окон редактора

Группа команд для разделения окон редактора насчитывает четыре команды:

- **Split Tab Group Vertically** (Разделить группу закладок вертикально);
- **Split Tab Group Horizontally** (Разделить группу закладок горизонтально);
- **Move to Opposite Tab Group** (Переместить в противоположную группу закладок);
- **Change Tab Group Orientation** (Изменить ориентацию группы закладок).

Более подробно эти команды будут описаны в *разделе 3.6*.

3.2.1.11. *Settings (Настройки)*

Команде соответствуют "горячие" клавиши: <Ctrl>+<Alt>+<S>.

Команда открывает диалог со всеми возможными настройками IntelliJ IDEA.

3.2.1.12. *Template Project Settings (Шаблон настроек проекта)*

Команда открывает диалог, с помощью которого можно задать настройки проекта "по умолчанию", т. е. те, что будут актуальны для вновь создаваемого проекта.

Подробнее об этих настройках написано в *разделе 4.2* и в *главе 5*.

3.2.1.13. *Save All (Сохранить все)*

"Горячие" клавиши: <Ctrl>+<S>.

Команда сохраняет все измененные документы, открытые в IntelliJ IDEA, включая настройки модулей и проектов.

3.2.1.14. *Export Settings (Экспорт настроек)*

Команда открывает диалог экспорта всех настроек IntelliJ IDEA (рис. 3.4), в котором можно выбрать те настройки, которые вы хотите экспортировать, а также место на диске, где должен быть создан файл с экспортированными настройками.

Рис. 3.4. Диалог экспорта настроек

В результате импорта в указанном месте создается JAR-файл, содержащий набор XML-файлов с настройками.

3.2.1.15. *Import Settings (Импорт настроек)*

Команда импортирования настройки IntelliJ IDEA из файла. Файл должен быть сохранен командой **Export Settings** (Экспорт настроек) (*см. раздел 3.2.1.14*).

3.2.1.16. *Synchronize (Синхронизировать)*

"Горячие" клавиши: <Ctrl>+<Alt>+<Y>.

Команда находит все файлы, которые были изменены извне, и перезагружает их. Обычно IntelliJ IDEA автоматически замечает изменившиеся файлы и перезагружает их, однако эта возможность доступна только при работе в операционных системах Windows и Linux. Так что команда иногда может оказаться полезной при работе с IntelliJ IDEA на других платформах, а также, если автоматическая синхронизация выключена (*см. раздел 4.1.1*).

3.2.1.17. *Reload from Disk (Перезагрузить с диска)*

Если некоторый файл был изменен внутри IntelliJ IDEA, но еще не был сохранен, с помощью этой команды можно загрузить с диска его предыдущую версию, при этом сделанные изменения будут утрачены.

3.2.1.18. Команды для работы с CVS

Группа команд для работы с CVS-репозиторием (Concurrent Version System — одна из версий системы контроля версий) содержит три команды:

- **Check Out from CVS** (Извлечь из CVS). Команда извлечения файлов из CVS (в данном случае термин "check out" обозначает специфическую операцию для CVS). В случае, если среди этих файлов существует хотя бы один проект IntelliJ IDEA (т. е. файл с расширением ipr), IntelliJ IDEA предложит открыть этот проект;
- **Import into CVS** (Импортировать в CVS). Команда позволяет импортировать существующий проект в CVS (в данном случае термин "import" также обозначает операцию, специфичную для CVS);
- **Browse CVS Repository** (Просмотр CVS-репозитория). Аналог просмотра файловой системы, только для CVS.

Данные команды, несмотря на то, что они присутствуют в меню изначально (с момента установки IntelliJ IDEA), не являются базовыми. Эти команды добавлены в меню благодаря специальному плагину, входящему в поставку IntelliJ IDEA — CVS Integration (интеграция с CVS). Более подробно работа с CVS-репозиторием будет рассмотрена в *разделе 12.3.2*, а работа с плагинами — в *главе 15*.

3.2.1.19. *Export to HTML (Экспорт в HTML)*

Команда позволяет экспортировать содержимое файла в HTML. Данная команда бывает очень полезной, если требуется опубликовать какие-либо исходные тексты на сайте в Интернете. При вызове этой команды открывается специальный диалог (рис. 3.5).

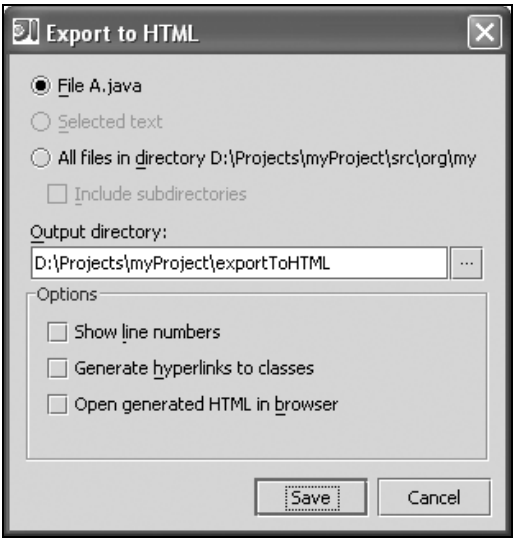


Рис. 3.5. Диалог экспорта в HTML

В диалоге можно задать следующие опции:

- объект экспорта:
  - текущий файл;
  - выделенный текст (эта опция активна, если в редакторе был выделен некоторый блок текста);
  - все файлы в директории (с возможностью включить все поддиректории, если они есть);
- путь на диске, куда должны быть записаны HTML-файлы;
- дополнительные настройки:
  - показывать номера строк (**Show line numbers**);
  - создавать гиперссылки на классы, которые встречаются в коде — в этом случае результатом экспорта будет полностью "кликабельная" (clickable) версия исходных файлов проекта с возможностью навигации между классами;
  - открыть сгенерированный HTML-код в браузере.

Самым главным достоинством данной операции является то, что сгенерированный HTML-код сохраняет все стилевые настройки, выделения и шрифты, имеющиеся в самой IntelliJ IDEA (*см. раздел 4.1.7*).

Кстати, при подготовке листингов к данной книге мы использовали именно эту функцию IntelliJ IDEA.

3.2.1.20. *Print (Печать)*

Команда вывода содержимого файла на печать. В случае вывода на печать также доступны расширенные настройки, показанные на рис. 3.7 и 3.8.

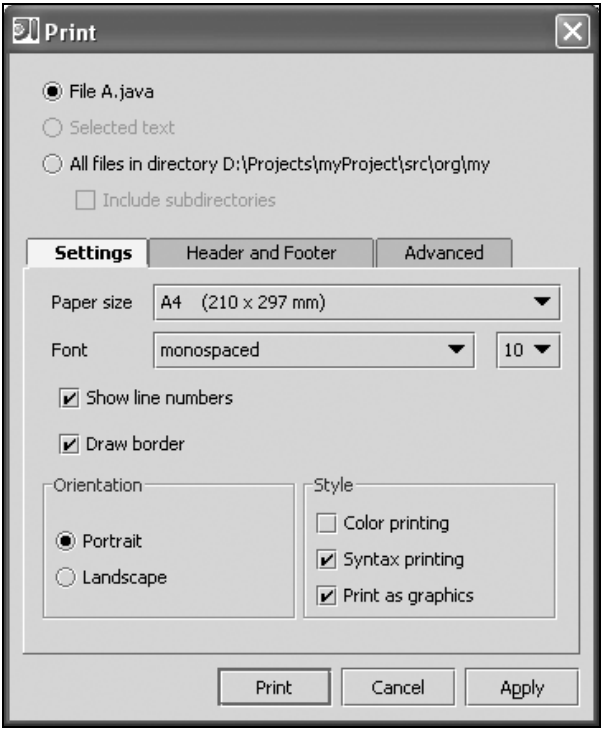


Рис. 3.6. Диалог вывода на печать: общие настройки

При активации команды вывода на печать появляется диалог, представленный на рис. 3.6.

Выбор содержимого для печати

Здесь так же, как и при экспорте в HTML, можно выбрать то, что будет напечатано:

- **File** (Файл) — отдельный указанный файл;
- **Selected text** (Выделенный текст) — блок текста, выделенный в редакторе IntelliJ IDEA;
- **All files in directory** (Все файлы в директории) — все файлы в указанной директории. Для печати файлов из указанной директории можно указать дополнительную опцию **Include subfolders** (Включить поддиректории), чтобы были напечатаны все файлы из всех вложенных директорий.

Закладка *Settings (Общие настройки)*

Общие настройки печати состоят из следующих опций:

- **Paper size** (Размер бумаги) — размер бумаги в принтере для печати;
- **Font** (Шрифт) — тип и размер шрифта;
- **Show line numbers** (Показывать номера строк) — выводить на печать также номера строк;
- **Draw border** (Рисовать рамку) — содержимое файлов, выводимых на печать, будет окружено рамкой;
- **Orientation** (Ориентация) — два варианта ориентации бумаги:
  - **Portrait** (Портрет) — портретная или книжная ориентация.
  - **Landscape** (Пейзаж) — пейзажная или альбомная ориентация.
- **Style** (Стиль):
  - **Color printing** (Цветная печать) — цветовые стили кода будут сохранены при выводе на печать;
  - **Syntax printing** (Печать синтаксиса) — стилевые выделения синтаксических конструкций кода будут сохранены при выводе на печать;
  - **Print as graphics** (Печать содержимого, как графики) — данная опция бывает полезна, если принтер плохо поддерживает печатаемые символы (например, в том шрифте, который вы задали). Также эту опцию можно использовать в случае, если принтер не поддерживает локальные символы, встречающиеся в комментариях.

Закладка *Header and Footer (Настройки "шапки" и "подвала")*

"Шапка" и "подвал" — это постоянные элементы, которые будут вставлены в начало и конец печатной страницы (см. рис. 3.7).

Настройка "шапки" и "подвала" довольно проста. Предполагается, что "шапка" и "подвал" могут состоять максимум из двух строк. При этом если в "шапке" уже есть одна строка, то "подвалу" остается тоже только одна строка. Имеется возможность настроить только две строки, которые попадают в "шапку" и/или "подвал".

Каждая строка имеет следующие настройки:

- **Text line** (Текстовая строка) — текст строки. Данная строка может содержать некоторые ключевые слова — макросы, используемые для подстановки некоторых значений. Всего есть два макроса:
  - **$FILE$** — название печатаемого файла;
  - **$PAGE$** — номер печатаемой страницы;

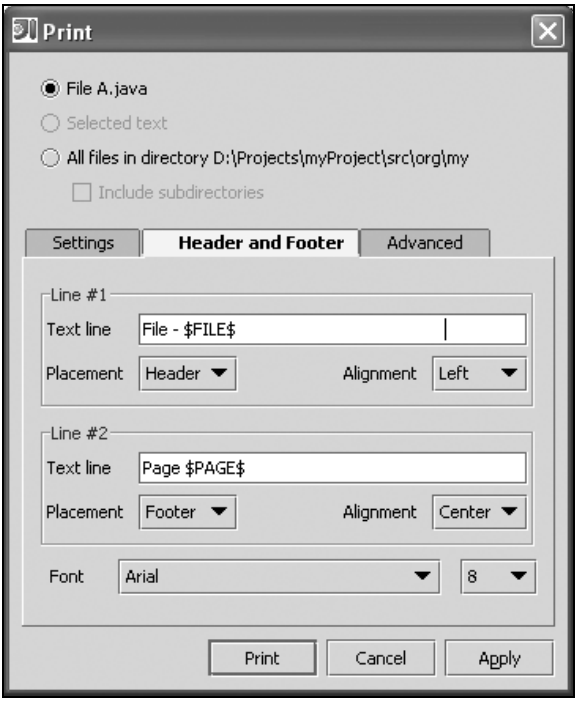


Рис. 3.7. Диалог вывода на печать: настройка "шапки" и "подвала"

- **Placement** (Размещение) — выпадающий список с двумя опциями:
  - **Header** ("Шапка") — размещать строку в "шапке";
  - **Footer** ("Подвал") — размещать строку в "подвале";
- **Alignment** (Выравнивание):
  - **Left** (Слева) — размещать строку с выравниванием по левому краю;
  - **Center** (По центру) — размещать строку по центру;
  - **Right** (По правому краю) — размещать строку с выравниванием по правому краю.

Для "шапки" и "подвала" также можно задать отдельные параметры шрифта (тип шрифта и его размер), которым они будут выводиться.

Закладка *Advanced (Дополнительные настройки печати)*

Дополнительные настройки печати представлены на рис. 3.8.

В качестве дополнительных настроек вывода на печать можно задать следующие опции:

- **Wrapping** (Перенос строк) — возможны два варианта переноса строк:
  - **No wrap** (Без переноса) — строки не переносятся;

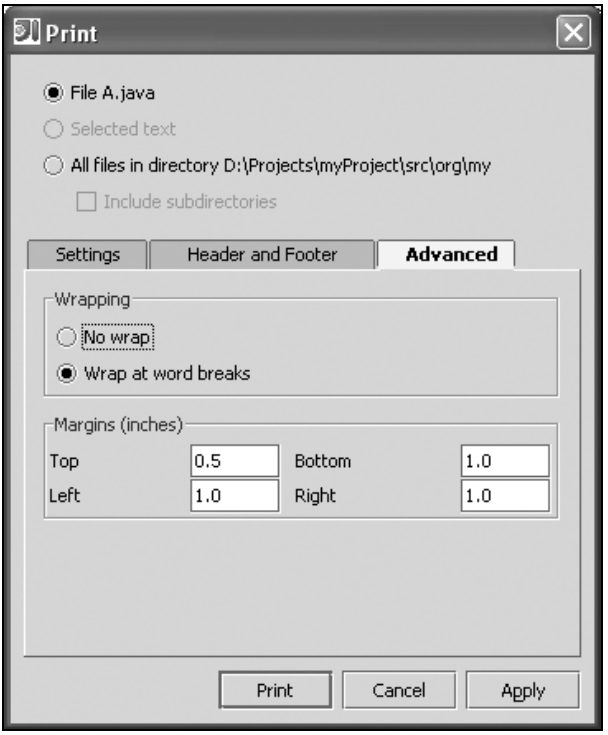


Рис. 3.8. Диалог вывода на печать: дополнительные настройки

- **Wrap at word breaks** (Перенос по разрывам строк) — IntelliJ IDEA будет делать переносы по разрывам в словах;

❒ **Margins** (Границы) — расстояние от края страницы до печатаемого содержимого;

- **Top** (Верх) — верхняя граница;
- **Left** (Слева) — левая граница;
- **Bottom** (Низ) — нижняя граница;
- **Right** (Право) — правая граница.

Размер отступа задается в дюймах.

3.2.1.21. *Exit (Выход)*

Данная команда завершает список команд в меню **File** (Файл). Она располагается последней в списке — там, где ей и положено находиться в уважающих себя приложениях ☺. Команда закрывает все открытые окна проектов и завершает работу IntelliJ IDEA.

3.2.2. Меню *Edit (Редактирование)*

Пункт меню **Edit** (Редактирование) содержит достаточно стандартный набор команд. Данные команды действуют как на текст в редакторе, так и на классы или файлы в инструментальных окнах **Project** (Проект) и **Commander** (Коммандер) (*см. разделы 3.4.8.2, 3.4.8.3*).

3.2.2.1. *Undo (Отменить)*

"Горячие" клавиши: <Ctrl>+<Z>.

Команда отмены последнего сделанного изменения. Под изменением понимается не только операция редактирования, но также и любая файловая операция, например создание или удаление файла.

3.2.2.2. *Redo (Повторить)*

"Горячие" клавиши: <Ctrl>+<Shift>+<Z>.

Команда повтора последнего изменения, отмененного с помощью команды **Undo** (Отменить) (*см. раздел 3.2.2.1*).

3.2.2.3. *Cut (Вырезать)*

"Горячие" клавиши: <Ctrl>+<X>.

Команда удаления в буфер обмена. Если в редакторе был выделен блок текста, удален будет именно он. Если выделения не было — удалена будет строка, в которой находится курсор. Вырезанная часть текста при этом попадает в буфер обмена.

Команда также действует на классы и файлы.

3.2.2.4. *Copy (Копировать)*

"Горячие" клавиши: <Ctrl>+<C>.

Команда копирования в буфер обмена. Если в редакторе был выделен блок текста, будет скопирован именно он. Если выделения не было — скопирована будет строка, в которой находится курсор.

Команда также действует на классы и файлы.

3.2.2.5. *Copy Path (Копировать путь)*

"Горячие" клавиши: <Ctrl>+<Shift>+<C>.

Команда копирует в буфер обмена путь текущего файла.

3.2.2.6. *Paste (Вставить)*

"Горячие" клавиши: <Ctrl>+<V>.

Команда вставки из буфера обмена. Команда действует как на текст в редакторе, так и на скопированные классы и файлы.

3.2.2.7. *Paste... (Вставить...)*

"Горячие" клавиши: <Ctrl>+<Shift>+<V>.

Команда расширенной вставки из буфера обмена, которая позволяет просмотреть все сделанные копирования и выбрать из них нужное (рис. 3.9). Команда действует только в редакторе.

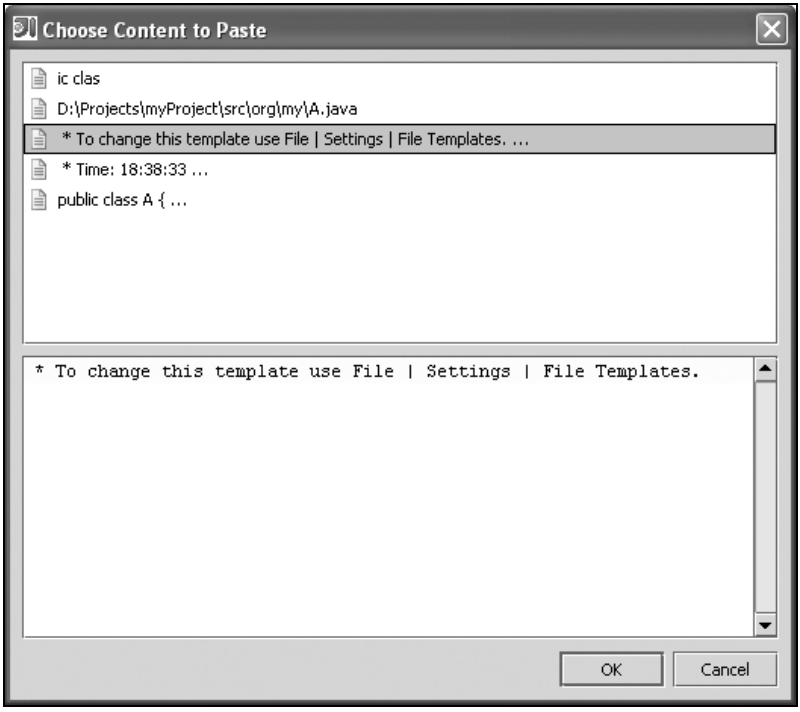


Рис. 3.9. Выбор содержимого буфера обмена для вставки

3.2.2.8. *Column Mode (Режим колонки)*

"Горячие" клавиши: <Ctrl>+<Shift>+<Insert>.

Команда переключения режима выделения со строчного (рис. 3.10) на колоночный с возможностью выделить вертикальный блок текста (рис. 3.11).

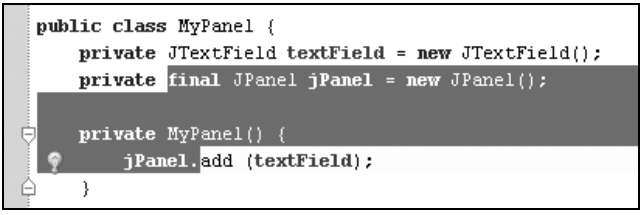


Рис. 3.10. Строчный режим выделения

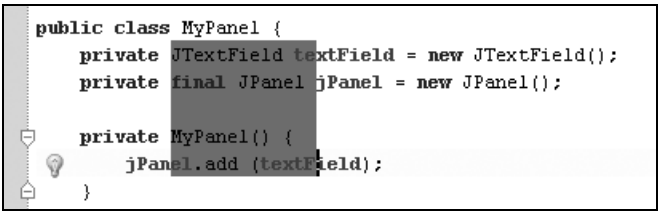


Рис. 3.11. Колоночный режим выделения

Команда действует при редактировании текста.

3.2.2.9. *Select All (Выделить все)*

"Горячие" клавиши: <Ctrl>+<A>.

Команда выделяет все в текущем контексте — текст или файлы.

3.2.2.10. *New (Создать)*

"Горячие" клавиши: <Alt>+<Insert>.

Данная команда позволяет создавать различные сущности. Тип сущностей, которые можно создать, зависит от контекста, в котором происходит создание.

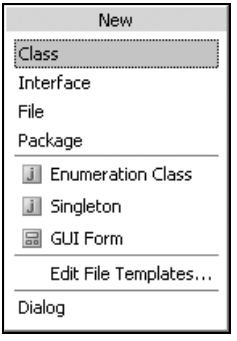


Рис. 3.12. Варианты создания сущностей в контекстном меню **Edit** | **New**

Например, поместив курсор в окне инструментов **Project** (Проект) на классе `A` (см. рис. 3.2) и вызвав команду **New** (Создать), мы увидим появившееся контекстное меню (рис. 3.12), в котором содержатся следующие пункты:

- **Class** (Класс) — создать класс, для которого необходимо ввести имя класса;
- **Interface** (Интерфейс) — создать интерфейс, для которого необходимо ввести имя интерфейса;
- **File** (Файл) — создать файл, для которого необходимо ввести имя файла;

- **Package** (Пакет) — создать пакет, для которого необходимо ввести имя пакета. Можно создавать пакеты разного уровня вложенности, для чего нужно ввести их названия, разделенные точкой, так, как это делается в Java (рис. 3.13). В результате работы такой команды будут созданы три вложенных директории com/mysite/myproject, соответствующие этим пакетам;

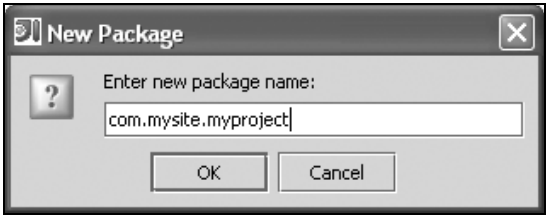


Рис. 3.13. Создание нового пакета

- **Enumeration Class** (Перечислимый класс) — создает класс, который представляет собой перечислимый набор значений (рис. 3.14). В результате получится класс, текст которого представлен в листинге 3.1.

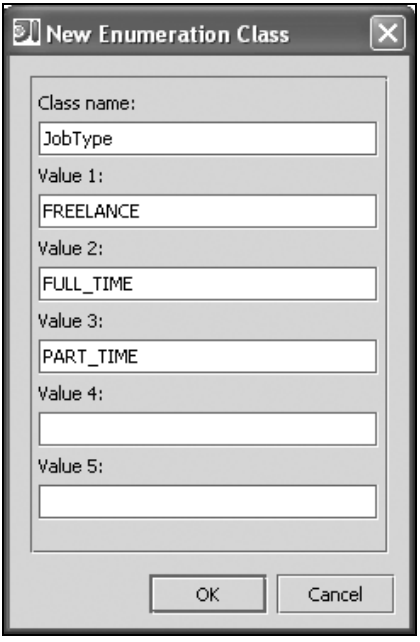


Рис. 3.14. Создание перечислимого класса

Листинг 3.1

```
public class JobType {
    public static final JobType FREELANCE = new JobType("FREELANCE");
    public static final JobType FULL_TIME = new JobType("FULL_TIME");
    public static final JobType PART_TIME = new JobType("PART_TIME");

    private final String myName; // for debug only

    private JobType(String name) {
        myName = name;
    }

    public String toString() {
        return myName;
    }
}
```

Примечание

Кстати, если вы настроили проект так, чтобы он использовал возможности Java 5 (*см. главу 11*), в контекстном меню появится несколько новых сущностей (рис. 3.15), в числе которых будет и новый тип данных `Enum`, который намного удобнее использовать, чем класс, представленный в листинге 3.1.

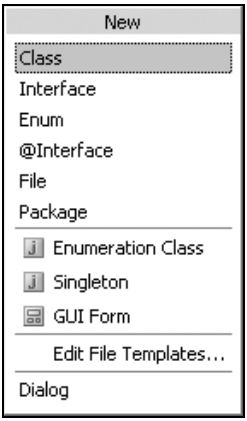


Рис. 3.15. Контекстное меню создания сущностей для проекта, использующего возможности языка Java 5

- **Singleton** (Синглетон) — создает класс, имеющий только один экземпляр при работе приложения. Для его создания необходимо ввести имя класса. Результат работы такой команды можно увидеть в листинге 3.2.

Листинг 3.2

```
public class MyManager {
    private static MyManager ourInstance = new MyManager();

    public static MyManager getInstance() {
        return ourInstance;
    }

    private MyManager() {
    }
}
```

- **GUI Form** (GUI — Graphical User Interface — форма графического интерфейса пользователя) — создает форму, которая может быть отредактирована во встроенном визуальном редакторе. Подробнее о создании форм можно прочитать в *главе 7*;
- **Edit File Templates** (Редактировать шаблоны файлов) — открывает диалог, в котором можно редактировать шаблоны файлов. Эта команда, по сути, дублирует одну из общих настроек IntelliJ IDEA, о которой подробнее можно прочитать в *разделе 4.1.14*;
- **Dialog** (Диалог) — данная команда создает визуальную форму и скелет класса для простейшего диалога — панели, на которой есть две кнопки: **OK** и **Cancel**.

Рис. 3.16. Создание нового диалога

При создании диалога можно указать несколько простых опций (рис. 3.16):

- Можно задать имя диалога — это имя будет использовано как для создаваемого класса, так и для формы;
- **Generate main() method that creates and shows dialog** (Создать метод `main()`, который создает и показывает диалог) — если установить эту опцию, в сгенерированном коде класса будет присутствовать соответствующий метод `main()`;
- **Generate 'OK 'handler** (Создать обработчик нажатия кнопки OK);
- **Generate 'Cancel' handler** (Создать обработчик нажатия кнопки Cancel).

Если ваш проект использует возможности языка Java 5, в контекстном меню создания появятся дополнительные пункты (см. рис. 3.15).

❐ **Enum** (Перечисление) — создает перечисляемый тип Java 5.

❐ **@Interface** (Аннотация интерфейса) — создает аннотацию интерфейса.

Набор сущностей можно самостоятельно расширять и модифицировать (*см. раздел 4.1.14*).

3.2.2.11. *Delete (Удалить)*

Команде соответствует клавиша <Delete>.

Команда удаления текста, класса или файла. При этом удаленный объект в буфер обмена не помещается — в этом отличие этой команды от команды **Cut** (Вырезать) (*см. раздел 3.2.2.3*).

3.2.2.12. *Toggle Bookmark (Поставить/Убрать закладку)*

Команде соответствует клавиша <F11>.

Данная команда позволяет поставить закладку в произвольном месте файла. Если повторить команду на строке, где уже стоит закладка — она будет снята. Визуально закладки отображаются с помощью галочки на полях слева от редактируемого файла, а также с помощью черной черты на полях справа (рис. 3.17).

Кстати, установить закладку можно и другим способом — достаточно лишь щелкнуть левой кнопкой мыши на полях одновременно с нажатой клавишей <Ctrl>.

3.2.2.13. *Show Bookmarks (Показать закладки)*

"Горячие" клавиши: <Shift>+<F11>.

Данная команда отображает диалог, в котором можно просмотреть и отредактировать закладки, сделанные во всех файлах проекта (рис. 3.18).

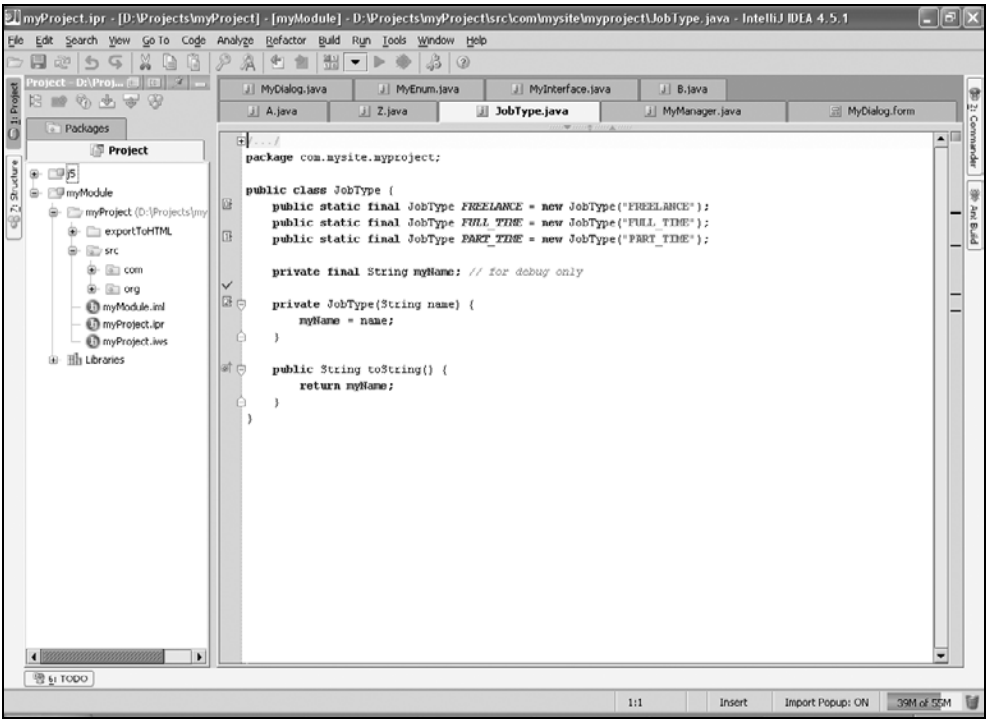


Рис. 3.17. Визуальное размещение закладок в окне редактора

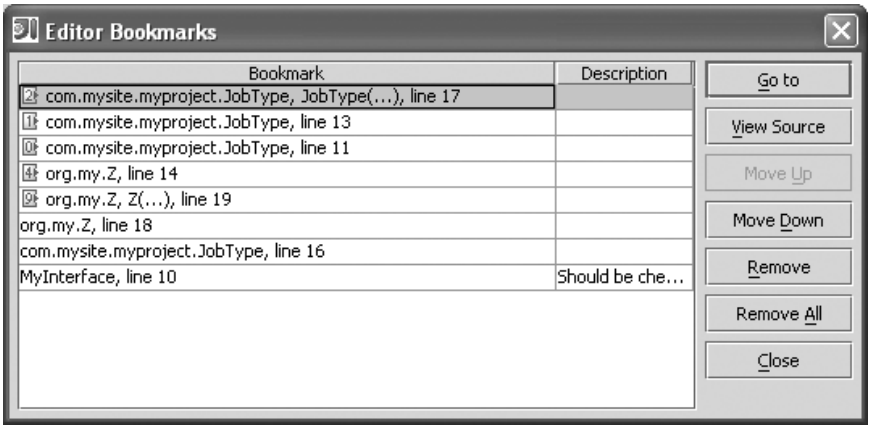


Рис. 3.18. Редактирование всех закладок, сделанных в файлах проекта

Кроме просмотра, данный диалог позволяет производить с закладками следующие действия:

- **Go to** (Перейти) — перейти к закладке — текущим окном редактора становится окно с выбранным файлом, а сам диалог будет закрыт;
- **View Source** (Просмотр исходного текста) — текущим окном редактора становится окно с выбранным файлом, но диалог не будет закрыт;
- **Move Up** (Поднять) — перемещает выбранную закладку на одну позицию вверх в списке закладок;
- **Move Down** (Опустить) — перемещает выбранную закладку на одну позицию вниз в списке закладок;
- **Remove** (Удалить) — удаляет закладку;
- **Remove All** (Удалить все) — удаляет все закладки.

Кроме простых безымянных закладок в IntelliJ IDEA существуют именованные, точнее пронумерованные — это закладки с порядковым номером от 0 до 9 (по числу цифровых клавиш). В общем меню нет команд, которыми можно эти закладки установить, тем не менее использовать их можно и даже нужно, потому что они намного удобнее, чем простые закладки.

Цифровые закладки устанавливаются с помощью сочетания клавиш <Ctrl>+<Shift> и цифровых клавиш <0>—<9>. Соответственно всего можно установить десять цифровых закладок. Визуально они отображаются там же, где и обычные закладки (см. рис. 3.17), но вместо галочки используют небольшую пиктограмму с цифрой.

Если попытаться установить закладку с номером, который уже используется в каком-либо файле, предыдущая закладка будет снята.

Просмотреть цифровые закладки можно с помощью той же самой команды **Show Bookmarks** (Показать закладки).

Для навигации по цифровым закладкам удобно использовать клавишу <Ctrl> в сочетании с цифровыми клавишами <0>—<9>.

3.2.3. Меню *Search (Поиск)*

IntelliJ IDEA обладает очень мощными возможностями поиска и замены текста.

3.2.3.1. *Find (Найти)*

"Горячие" клавиши: <Ctrl>+<F>.

Команда поиска в тексте. В IntelliJ IDEA простой поиск начинается с диалога (рис. 3.19), в котором нужно ввести строку для поиска.

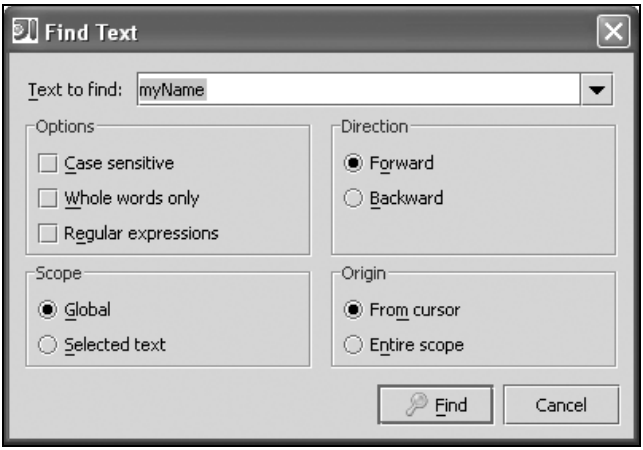


Рис. 3.19. Диалог поиска текста

Поиск можно расширять и сужать, используя следующие настройки:

- **Options** (Опции поиска):
  - **Case sensitive** (Чувствительный к регистру букв);
  - **Whole words only** (Слова только целиком) — данная опция позволяет исключить из результатов поиска фрагменты слов, соответствующие искомой строке. То есть если, например, искать слово "public", то определение класса "public class A" найдено не будет;
  - **Regular expressions** (Регулярные выражения) — воспринимать строку поиска как регулярное выражение. Синтаксис регулярного выражения соответствует `java.util.regex`;
- **Direction** (Направление поиска):
  - **Forward** (В прямом порядке) — поиск от курсора к концу документа;
  - **Backward** (В обратном порядке) — поиск от курсора к началу документа;
- **Scope** (Область поиска):
  - **Global** (Глобально) — поиск во всем тексте;
  - **Selected text** (Выделенный текст) — поиск только в выделенном блоке текста;
- **Origin** (Начало) — откуда начинать поиск:
  - **From cursor** (С курсора) — начинать поиск с позиции курсора;
  - **Entire scope** (С начала области поиска) — начинать поиск с первой позиции в тексте.

Если поиск прошел успешно, найденный фрагмент текста будет выделен.

3.2.3.2. *Replace (Заменить)*

"Горячие" клавиши: <Ctrl>+<R>.

Команда замены текста похожа на команду поиска в тексте. Она также использует все параметры поиска. Однако она заменит найденные фрагменты текста строкой, которую вы ввели в поле ввода **Replace with** (Заменить на) диалога замены текста (рис. 3.20).

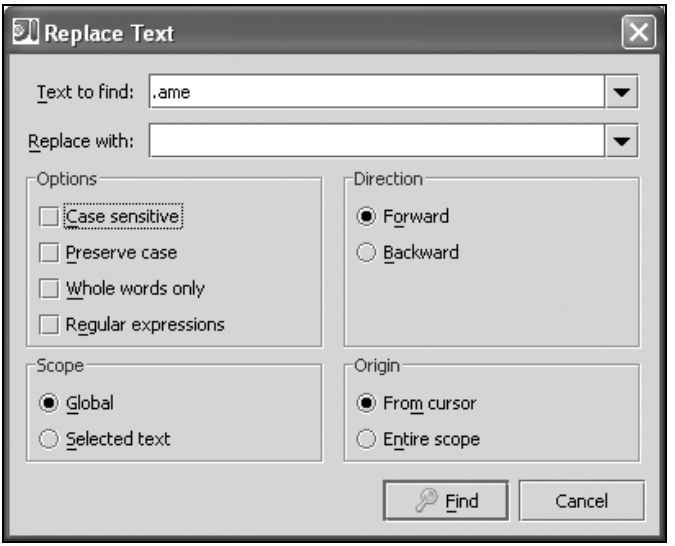


Рис. 3.20. Диалог замены текста

Как можно заметить на рис. 3.20, по сравнению с диалогом поиска, добавилась одна дополнительная опция **Preserve case** (Сохранять регистр букв) — она позволяет сохранять регистр букв в заменяемых фрагментах. Например, если вы выполняете команду замены слова "remote" на "local" и установили эту опцию, то в тексте все вхождения "Remote" будут заменены "Local", т. е. регистр символов останется таким, каким он был в оригинале.

Если в результате поиска будет найден хотя бы один соответствующий фрагмент текста, IntelliJ IDEA предложит заменить его.

3.2.3.3. *Find Next (Искать дальше)*

"Горячие" клавиши: <F3>.

Команда продолжения поиска. Если будет найден еще один фрагмент текста, он будет подсвечен. Данная команда актуальна как для поиска, так и для замены текста.

3.2.3.4. *Find Previous (Найти предыдущее)*

"Горячие" клавиши: <Shift>+<F3>.

Команда продолжения поиска в обратном направлении, т. е. если команда поиска или замены нашла несколько фрагментов текста. С помощью этой команды можно перейти к предыдущему найденному фрагменту.

3.2.3.5. *Find Word at Caret (Найти слово под курсором)*

"Горячие" клавиши: <Ctrl>+<F3>.

Команда считывает слово, на котором в данный момент стоит курсор, и производит его поиск. При этом используются настройки поиска по умолчанию, т. е. поиск будет осуществлен в прямом направлении (**Direction**: **Forward**) по всему тексту (**Scope**: **Global**), начиная с текущей позиции курсора (**Origin**: **From cursor**).

3.2.3.6. *Incremental Search (Инкрементальный поиск)*

"Горячие" клавиши: <Alt>+<F3>.

При вызове данной команды в верхней части окна редактора появляется небольшое поле, похожее на всплывающую подсказку (рис. 3.21).

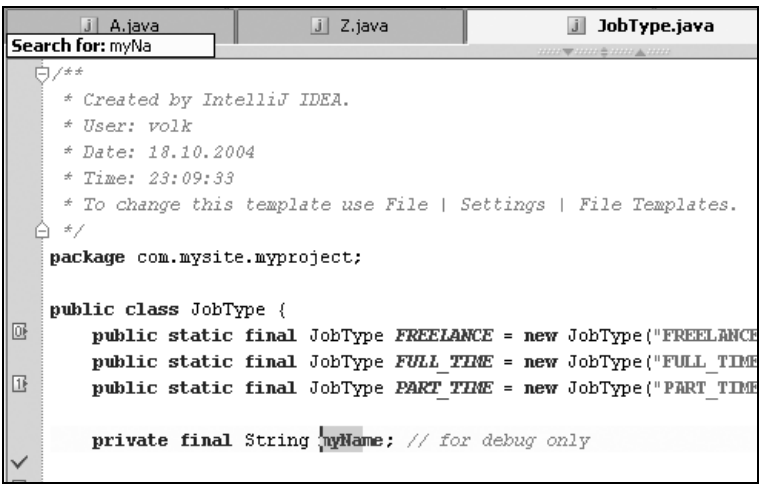


Рис. 3.21. Инкрементальный поиск

По мере набора на клавиатуре строки, которую вы хотите найти, она будет появляться в этом поле, а подсветка внутри редактора сместится на первый найденный фрагмент текста. Затем, когда вы ввели всю строку, можно нажимать клавиши <↑> (курсор вверх) и <↓> (курсор вниз), чтобы перемещаться вперед-назад между найденными фрагментами. Для завершения поиска дос-

таточно нажать клавишу <Escape> или щелкнуть мышью в произвольное место в тексте.

По скорости вызова и удобству использования инкрементальный поиск превосходит обычный поиск, поэтому им очень удобно пользоваться по мере написания кода (экономится как минимум два нажатия клавиш). Инкрементальный поиск работает не только в Java-файлах, но и в любых других файлах, поддерживаемых IntelliJ IDEA.

3.2.3.7. *Find Usages (Поиск обращений)*

"Горячие" клавиши: <Alt>+<F7>.

Мощная команда поиска для Java-кода внутри файлов, находящихся в проекте. Поиск может производиться:

- в файлах с исходными текстами на Java;
- в скомпилированных классах, в том числе и находящихся в библиотеках;
- в файлах JSP.

Можно производить поиск обращений к следующим составляющим Java-кода:

- пакетам;
- классам;
- методам;
- переменным — полям класса, параметрам вызова метода, локальным переменным.

Далее мы подробно рассмотрим каждую из этих возможностей поиска.

Для всех типов поиска обращений имеются общие настройки, позволяющие сузить или расширить область поиска, а также опции, указывающие на способ отображения результатов поиска:

- **Search in non-java files** (Искать не только в Java-файлах) — опция позволяет включить в область поиска не только Java-файлы;
- **Skip results tab with one usage** (Не открывать результаты поиска, если найден только один вариант) — данная опция позволяет пропустить вывод промежуточных результатов в инструментальном окне **Find** (Поиск) (*см. раздел 3.4.8.4*) и сразу перейти к месту использования искомого символа в коде;
- **Scope** (Границы поиска): настройка границ поиска:
  - **Project Classes** (Классы проекта) — классы проекта, включая классы тестов;
  - **Project Production Classes** (Основные классы проекта) — классы проекта без классов тестов;

- **Project Test Classes** (Классы тестов проекта) — только классы тестов проекта;
- **Current File** (Текущий файл) — данное значение границ поиска позволяет сузить область поиска до текущего файла. В IntelliJ IDEA имеется отдельная команда для поиска в текущем файле (*см. раздел 3.2.3.8*);
- **Previous Search result** (Результаты предыдущего поиска) — данное значение доступно, только если вы уже производили поиск обращений. В этом случае новый поиск будет осуществлен среди результатов, найденных предыдущим поиском;
- **Class Hierarchy** (Иерархия класса) — производить поиск в иерархии текущего класса.

❐ **Open in new tab** (Открыть в новой закладке) — опция позволяет открыть результаты поиска в отдельной закладке инструментального окна **Find** (Поиск) (*см. раздел 3.4.8.4*).

Поиск обращения к пакетам

Для поиска обращения к пакетам необходимо подвести курсор к названию пакета в коде, либо в одном из инструментальных окон и вызвать команду **Find Usages** (Поиск обращений) (рис. 3.22).

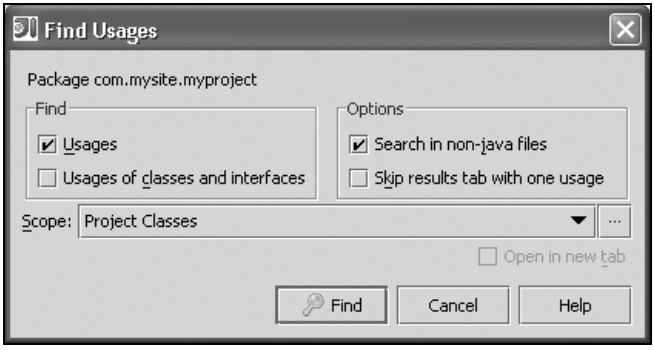


Рис. 3.22. Поиск обращений к пакетам

Для данного типа поиска можно задать дополнительные опции:

❐ **Usages** (Обращения) — искать обращения к пакету. Чтобы произвести поиск, данная опция должна быть включена;

❐ **Usages of classes and interfaces** (Обращения к классам и интерфейсам) — опция позволяет дополнительно находить места в коде, где используются классы и интерфейсы, расположенные в пакете.

Результат поиска выводится в инструментальном окне **Find** (Поиск) (рис. 3.23).

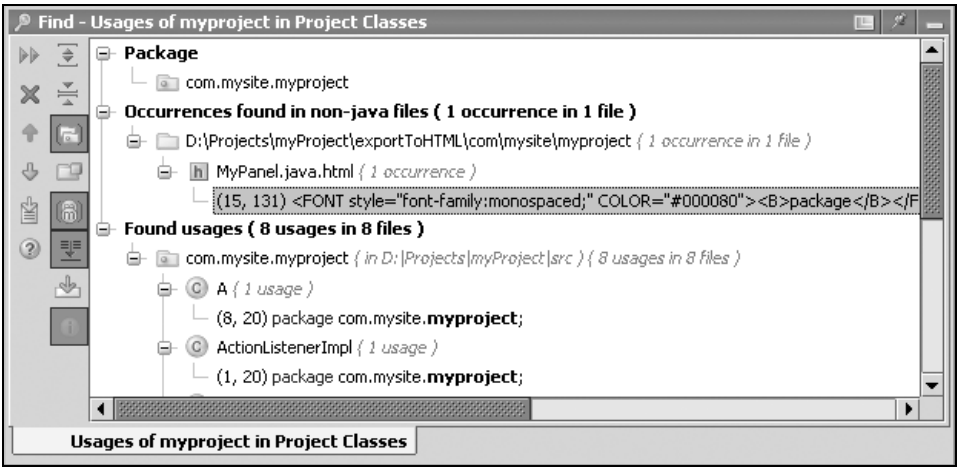


Рис. 3.23. Результаты поиска пакета com.mysite.myproject в инструментальном окне **Find**

Найденные результаты отображаются в сгруппированном виде. Группировку можно менять с помощью кнопок в панели инструментов:

- **Group by packages** (Группировать по пакетам) <Ctrl>+<P>;
- **Group by modules** (Группировать по модулям) <Ctrl>+<D>;
- **Group by methods** (Группировать по методам) <Ctrl>+<M>.

Поиск обращения к классам

Для поиска обращения к классу необходимо выполнить команду **Find Usages** (Поиск обращений), подведя курсор к имени класса в коде или в одном из инструментальных окон (рис. 3.24).

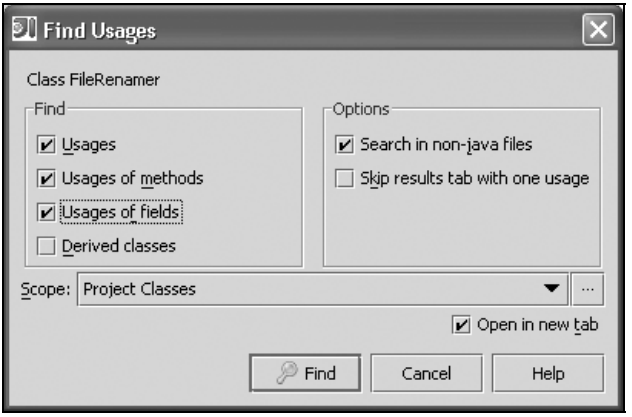


Рис. 3.24. Поиск обращений к классу

Для данного типа поиска можно задать дополнительные опции:

- **Usages** (Обращения) — искать обращения к классу. Чтобы произвести поиск, данная опция должна быть включена;
- **Usages of methods** (Обращения к методам) — искать дополнительно обращения к методам класса;
- **Usages of fields** (Обращения к полям класса) — искать дополнительно обращения к полям класса;
- **Derived classes** (Производные классы) — искать дополнительно производные классы, в случае поиска обращений к классу;
- **Implemented classes** (Имплементирующие классы) — искать дополнительно среди классов, имплементирующих искомый интерфейс, в случае поиска обращений к интерфейсу.

Результаты поиска класса также отображаются в инструментальном окне **Find** (Найти), аналогичном тому, что представлено на рис. 3.23.

Поиск обращения к полю класса

Для поиска обращений к полю класса необходимо выполнить команду **Find Usages** (Поиск обращений), подведя курсор к имени поля класса в коде или в одном из инструментальных окон (рис. 3.25).

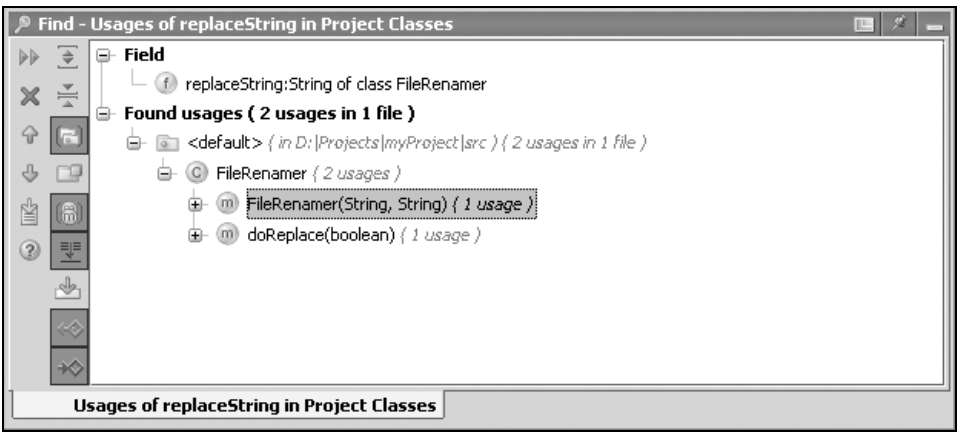


Рис. 3.25. Результаты поиска поля класса в инструментальном окне **Find**

Если поле класса имеет методы `get` или `set`, IntelliJ IDEA задаст дополнительный вопрос о поиске обращений среди этих методов (рис. 3.26).

Дополнительно к имеющимся кнопкам на панели инструментов окна **Find** (Найти) при данном типе поиска добавилось две дополнительных, с помощью которых можно сократить количество отображаемых результатов:

- **Show read access** (Доступ на чтение) <Ctrl>+<R> — показать только те обращения к полю класса, которые имеют к нему доступ только на чтение;

- **Show write access** (Доступ на запись) <Ctrl>+<W> — показать только те обращения к полю класса, которые имеют к нему доступ только на запись.

Рис. 3.26. Дополнительный вопрос о поиске методов `get` и `set` при поиске поля класса

Поиск параметра вызова метода и локальной переменной

Данный тип поиска практически не отличается от поиска поля класса. В случае, если поиск производится для параметра метода, IntelliJ IDEA задаст дополнительный вопрос о поиске обращения к данному параметру в методах, переопределяющих данный (рис. 3.27).

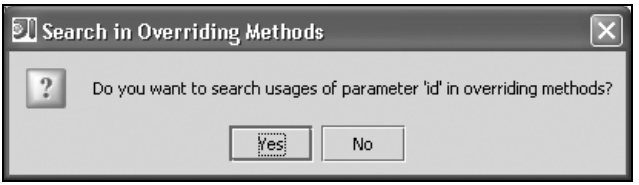


Рис. 3.27. Дополнительный вопрос о поиске обращения к параметру метода в переопределениях метода

3.2.3.8. *Find Usages in File (Поиск обращений в файле)*

"Горячие" клавиши: <Ctrl>+<F7>.

Данная команда представляет собой сокращенный вариант команды **Find usages** (Поиск обращений) (*см. раздел 3.2.3.7*) с областью поиска, ограниченной текущим файлом.

3.2.3.9. *Highlight Usages in File (Подсветить обращения в файле)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F7>.

Данный тип поиска представляет собой смесь из обычного текстового поиска в файле командой **Find** (Поиск) (*см. раздел. 3.2.3.1*) и поиска обращений в файле командой **Find Usages in File** (Поиск обращений в файле) (*см. раздел 3.2.3.8*). Результат поиска данной команды имеет другое визуальное

представление: все найденные фрагменты подсвечиваются прямо в тексте, а также помечаются с помощью специальных отметок на полях редактора.

Команда поддерживает поиск с распознаванием структуры Java-кода, а также обычный текстовый поиск.

Первый вариант поиска работает только в Java-коде. Для того чтобы он заработал, необходимо подвести курсор к какому-либо элементу Java-кода и вызвать команду. Распознаются следующие элементы:

- ❒ классы и интерфейсы;
- ❒ методы;
- ❒ поля классов;
- ❒ переменные.

Если такие элементы были найдены, они будут подсвечены в тексте и отмечены черточками на правых полях окна редактора (рис. 3.28).

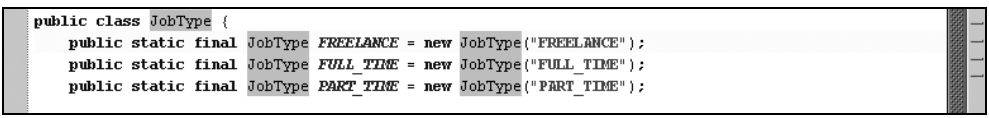


Рис. 3.28. Результат работы команды подсветки обращений

Второй вариант поиска — полнотекстовый — не распознает, чем являлся найденный фрагмент, а воспринимает его как обычный текст. Визуально результат работы такого поиска аналогичен первому варианту, за исключением цвета подсветки и, возможно, количеству найденных фрагментов.

После того как были найдены и подсвечены фрагменты текста, между ними можно перемещаться с помощью команд **Find Next** (Найти следующее) и **Find Previous** (Найти предыдущее) (*см. разделы 3.2.3.3, 3.2.3.4*).

Чтобы подсветка исчезла, нужно нажать клавишу <Escape>.

Если поиск производился по имени переменной, найденные фрагменты будут подсвечены разным цветом в зависимости от того, какого рода обращение к переменной встречается в коде. Если переменной присваивается значение — выделение будет малиновым, если считывается ее значение — фиолетовым, в прочих найденных фрагментах (которые не являются Java кодом) выделение будет зеленым.

3.2.3.10. *Find in Path (Поиск в пути)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F>.

Данная команда производит поиск текста среди файлов всего проекта, отдельного модуля или указанной директории (рис. 3.29). Также можно задать дополнительный фильтр имени файла, чтобы ограничить охват поиска.

Фильтр имени имеет стандартный вид и понимает стандартные символы масок файлов операционной системы:

- ? — любой символ;
- * — группа символов.

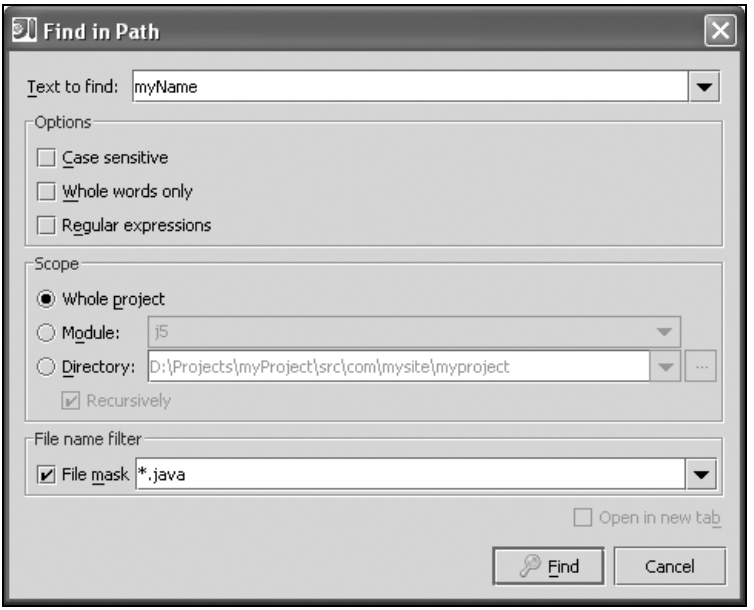


Рис. 3.29. Поиск в пути

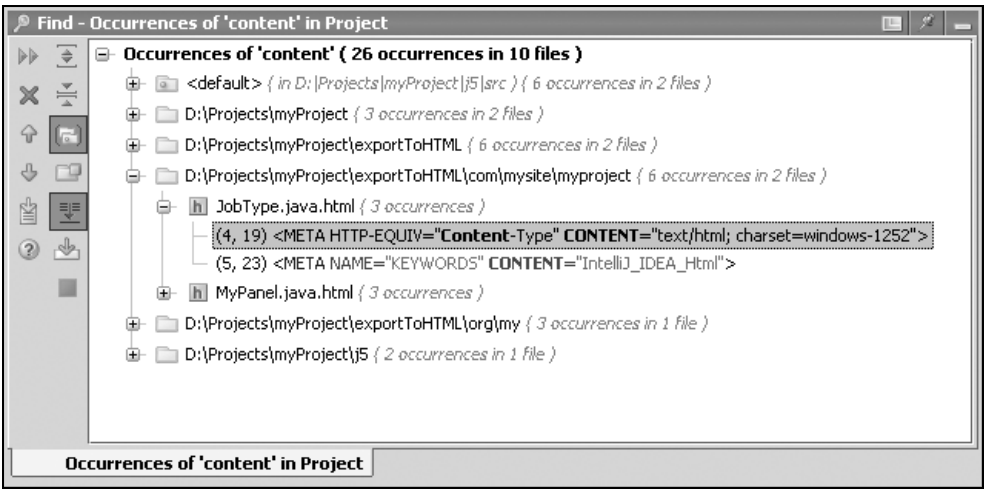


Рис. 3.30. Результаты поиска в пути

Так как найденных фрагментов текста может быть очень много, результаты поиска отображаются в отдельном инструментальном окне **Find** (Найти), показанном на рис. 3.30.

Подробнее о возможностях данного инструментального окна *см. раздел 3.4.8.4.*

3.2.3.11. *Replace in Path (Замена в пути)*

"Горячие" клавиши: <Ctrl>+<Shift>+<R>.

Данная команда аналогична команде **Find in Path** (Поиск в пути) с той лишь разницей, что она кроме поиска также производит и замену текста.

После выполнения данной команды открывается инструментальное окно **Find** (Найти), закладка найденных фрагментов текста которого дополнена тремя кнопками (рис. 3.31):

- **Do Replace All** (Заменить все) — выполнить замену текста во всех найденных фрагментах;
- **Replace Selected** (Заменить выделенные) — выполнить замену только для выделенных фрагментов;
- **Cancel** (Отменить) — отменить замену.

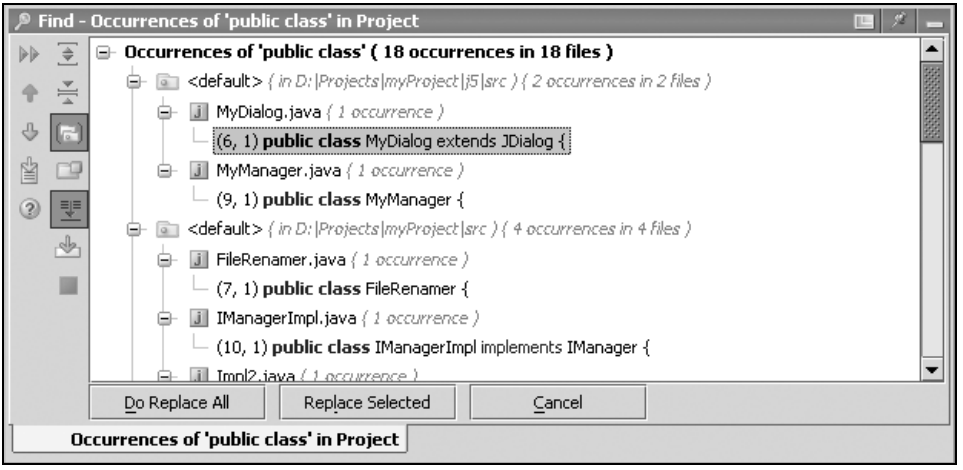


Рис. 3.31. Результаты поиска фрагментов для замены команды **Replace in Path**

3.2.3.12. Структурный поиск и замена

Структурный поиск — относительно новая функция IntelliJ IDEA (впервые появилась в версии 4.5). Функция структурного поиска позволяет задать маску поиска в виде параметризованной структуры Java-кода.

В меню **Search** (Поиск) функции структурного поиска представлены двумя командами:

- **Search Structurally** (Структурный поиск) — структурный поиск Java-кода;
- **Replace Structurally** (Структурная замена) — структурная замена Java-кода.

Подробнее структурный поиск будет рассмотрен в *разделе 6.5*.

3.2.4. Меню *View (Просмотр)*

Данный раздел меню содержит команды для отображения отдельных элементов интерфейса IntelliJ IDEA, а также для просмотра различной дополнительной информации.

3.2.4.1. *Toolbar (Панель инструментов)*

Команда-переключатель отображения панели инструментов. По умолчанию панель инструментов включена.

3.2.4.2. *Status Bar (Панель статуса)*

Команда-переключатель панели статуса. По умолчанию панель статуса включена.

3.2.4.3. *Show Whitespaces (Показывать пробелы)*

Команда-переключатель визуального отображения пробельных символов (табуляций и самих пробелов). По умолчанию пробельные символы выключены. Если переключатель включен, пробелы в тексте отображаются маленькими точками, а табуляции стрелочками (рис. 3.32, 4.22).

```
public class JobType {
    public static final JobType FREELANCE = new JobType("FREELANCE");
    public static final JobType FULL_TIME = new JobType("FULL_TIME");
    public static final JobType PART_TIME = new JobType("PART_TIME");

    private final String myName; // for debug only
```

Рис. 3.32. Отображение пробелов с помощью маленьких точек

Обратите внимание на пятую строчку на рис. 3.32. Если бы исходный Java-код изначально разрабатывался в IntelliJ IDEA, в ней бы не было столько отображенных пробелов в пустой строке ☺.

3.2.4.4. *Full Screen (Полный экран)*

"Горячие" клавиши: <Ctrl>+<Alt>+<F11>.

Команда переключения в полноэкранный режим и обратно.

3.2.4.5. *File Structure Popup (Всплывающее окно структуры файла)*

"Горячие" клавиши: <Ctrl>+<F12>.

Данная команда отображает модальный диалог со списком методов и полей класса (рис. 3.33).

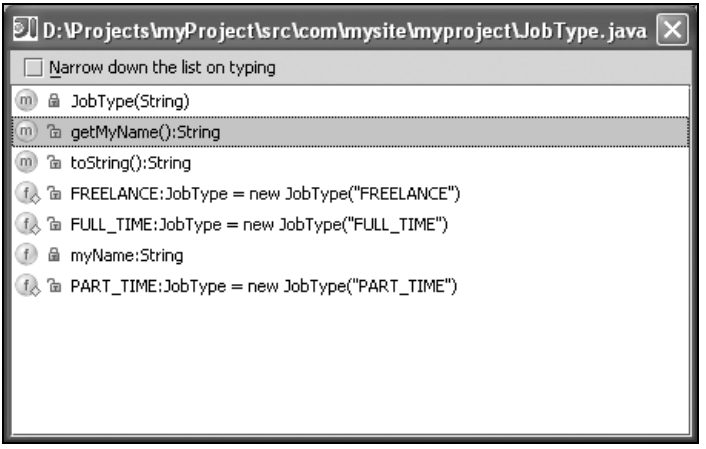


Рис. 3.33. Всплывающее окно структуры файла

То, что отображается в этом диалоге, очень похоже на содержимое инструментального окна **Structure** (Структура) (*см. раздел 3.4.8.8*). Однако данный диалог отображает структуру в более простом виде. Его удобнее использовать для быстрой навигации по содержимому файла.

В данном окне работает функция инкрементального поиска (*см. раздел 3.2.3.6*).

Опция **Narrow down the list on typing** (Сокращать список по мере печати) позволяет автоматически сокращать список элементов файла, удовлетворяющих набранной строке для поиска, что очень удобно при просмотре файлов, содержащих большой объем кода.

3.2.4.6. *Select In (Отобразить в...)*

"Горячие" клавиши: <Alt>+<F1>.

Данная команда открывает всплывающее меню, в котором перечислены все возможные места (дополнительные инструментальные окна), где можно отобразить (найти или подсветить) выбранный файл. Для Java-файлов таких мест четыре:

- **Project** (Проект) — закладка **Project** (Проект) инструментального окна **Project** (Проект) (*см. раздел 3.4.8.2*);

- **File Structure** (Структура файла) — инструментальное окно **File Structure** (Структура файла) (*см. раздел 3.4.8.8*);
- **Packages** (Пакеты) — закладка **Packages** (Пакеты) инструментального окна **Project** (Проект) (*см. раздел 3.4.8.2*);
- **Commander** (Коммандер[1]) — инструментальное окно **Commander** (Коммандер) (*см. раздел 3.4.8.3*).

Для XML-файлов отсутствует пункт **Packages** (Пакеты), а для прочих файлов также и пункт **File Structure** (Структура файла).

Если выбрать один из вариантов, откроется соответствующее инструментальное окно, в котором будет отображен текущий файл.

3.2.4.7. *Quick JavaDoc (Быстрый JavaDoc)*

"Горячие" клавиши: <Ctrl>+<Q>.

Команда вызывает всплывающее окно, в котором отображается JavaDoc для выбранного элемента (рис. 3.34).

На рис. 3.34 мы выполнили команду **Quick JavaDoc** (Быстрый JavaDoc), установив курсор на тип возвращаемого параметра `String` метода `toString()`.

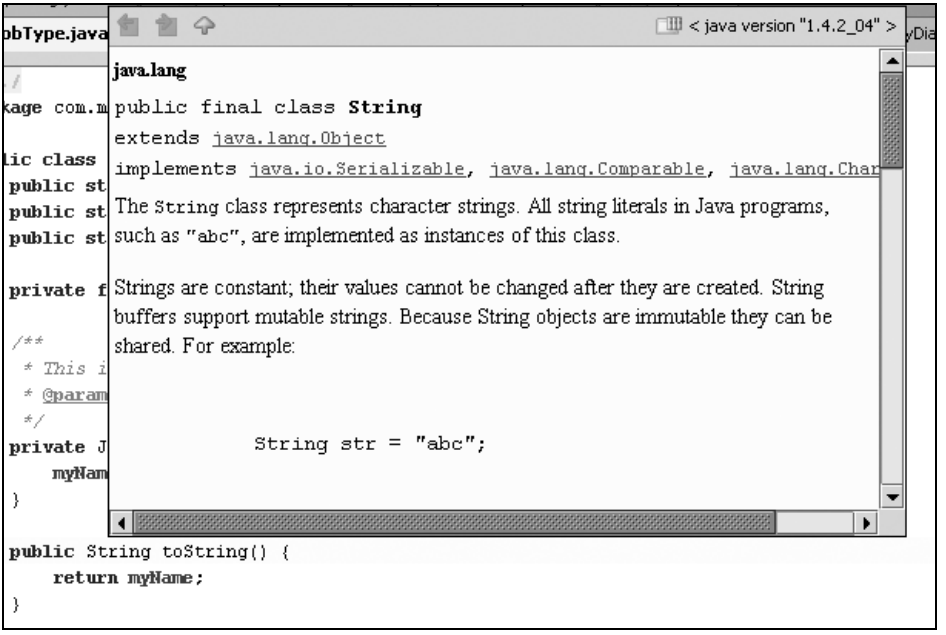


Рис. 3.34. Результат работы команды **Quick JavaDoc**

[1] Читается с ударением на "а", по аналогии с названием популярного файлового менеджера Norton Commander.

3.2.4.8. *Parameter Info (Информация о параметрах)*

"Горячие" клавиши: <Ctrl>+<P>.

Команда отображает информацию о параметрах выбранного метода во всплывающем окне (рис. 3.35).

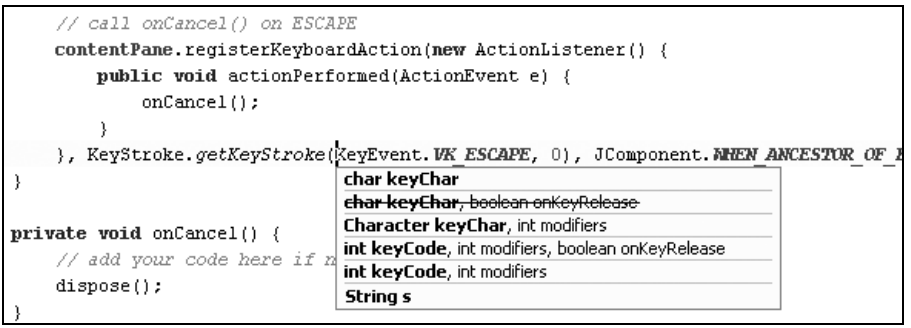


Рис. 3.35. Результат работы команды **Parameter Info**

Если выбранный метод имеет несколько вариантов передачи параметров (например, так, как это представлено на рис. 3.35), будет выведен список из всех возможных вариантов.

Обратите внимание на параметры, перечеркнутые линией. Это означает, что данное использование метода устарело (имеет аннотацию `@deprecated`).

3.2.4.9. *Context Info (Информация о контексте)*

"Горячие" клавиши: <Alt>+<Q>.

Данная команда отображает ближайшее невидимое начало структурного блока, к которому относится выбранная позиция курсора в файле (рис. 3.36).

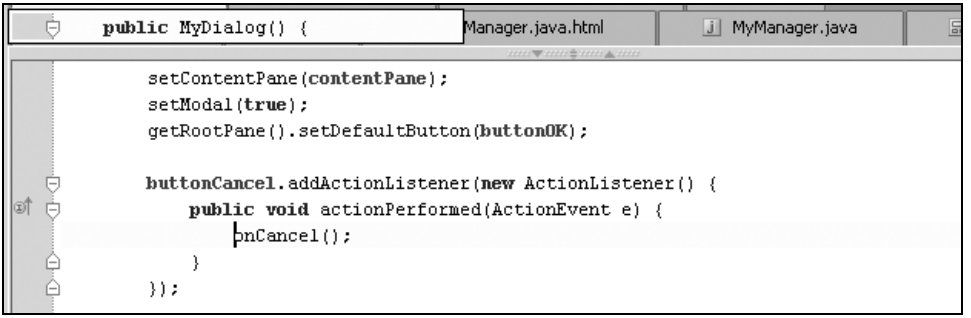


Рис. 3.36. Результат работы команды **Context Info**

Блок кода, в котором находится курсор на рис. 3.36, располагается внутри конструктора `MyDialog()`, что и отображено во всплывающем окне с информацией о контексте.

Если немного прокрутить текст вверх в окне редактора, так, чтобы скрылось начало создания имплементации `ActionListener`, и вызвать информацию о контексте, информация во всплывающем окне изменится (рис. 3.37).

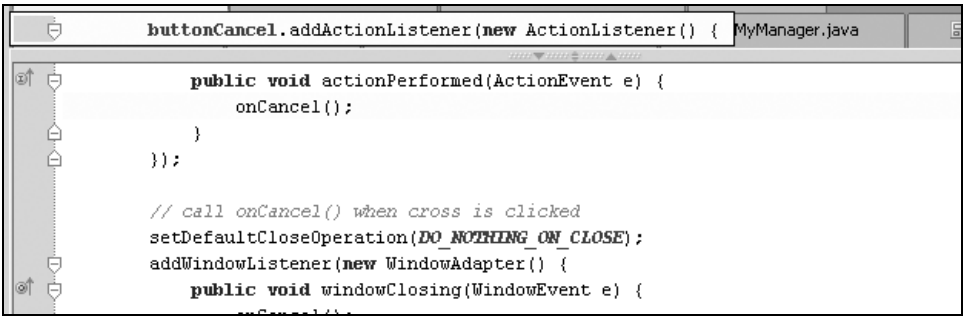


Рис. 3.37. Результат работы команды **Context Info**

3.2.4.10. *Error Description (Описание ошибки)*

"Горячие" клавиши: <Ctrl>+<F1>.

Команда отображает всплывающее окно, в котором содержится описание ошибки в коде, где располагается курсор (рис. 3.38).

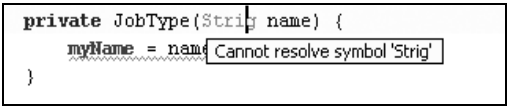


Рис. 3.38. Результат работы команды **Error Description**

В названии типа `String` есть опечатка (написано `Strig`), которую и нашла IntelliJ IDEA: "`Cannot resolve symbol Strig`" (Не могу определить символ `Strig`).

3.2.4.11. *Jump to Source (Перейти к исходному тексту)*

Команде соответствует клавиша <F4>.

Команда открывает в редакторе файл, который соответствует выбранной сущности в одном из инструментальных окон.

3.2.4.12. *Show Source (Показать исходный текст)*

"Горячие" клавиши: <Ctrl>+<Enter>.

Команда аналогичная команде **Jump to Source** (Перейти к исходному тексту) (*см. раздел 3.2.4.11*), но работающая с небольшим отличием. При выполнении этой команды фокус ввода не перемещается в открывшееся окно редактирования, а остается там, где была активирована команда (т. е. в одном из инструментальных окон).

3.2.4.13. *External JavaDoc (Внешний JavaDoc)*

"Горячие" клавиши: <Shift>+<F1>.

Данная команда аналогична команде **Quick JavaDoc** (Быстрый JavaDoc) (*см. раздел 3.2.4.7*) за тем исключением, что JavaDoc открывается в отдельном окне браузера (о настройках браузера дополнительно написано в *разделе 4.1.1*). Для выполнения этой команды необходимо, чтобы у интересующего вас кода были сконфигурированы пути для JavaDoc (*см. разделы 5.4.4, 4.1.8*).

3.2.4.14. *Recent Files (Последние файлы)*

"Горячие" клавиши: <Ctrl>+<E>.

Данная команда отображает список недавно использованных файлов, а также тех файлов, что открыты в редакторе (рис. 3.39).

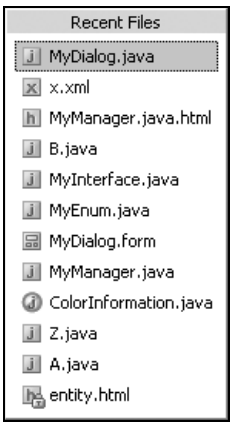


Рис. 3.39. Список последних использованных файлов

В списке последних файлов работает инкрементальный поиск (*см. раздел 3.2.3.6*).

3.2.4.15. *Quick Switch Scheme (Быстрая смена схемы)*

"Горячие" клавиши: <Ctrl>+<Back Quote>.

При активации данной команды отображается список схем для быстрой смены.

Рис. 3.40. Список схем для быстрой смены

Всего есть четыре варианта схем, которые можно быстро сменить (рис. 3.40):

1. **Switch Color Scheme** (Изменить цветовую схему) (*см. раздел 4.1.6*).
2. **Switch Code Style Scheme** (Изменить схему стилей кода) (*см. раздел 4.1.7*).
3. **Switch Keymap** (Изменить схему "горячих" клавиш) (*см. раздел 4.1.15*).
4. **Switch LookFeel** (Изменить схему отображения Look&Feel) (*см. раздел 4.1.2*).

Для примера, если выберем четвертый элемент в списке, мы увидим еще один список из доступных Look&Feel (рис. 3.41).

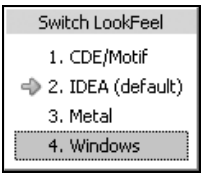


Рис. 3.41. Список доступных вариантов для смены схемы Look&Feel

Стрелкой отмечен текущий Look&Feel, который мы можем сменить на любой другой из списка.

Данная команда дублирует функциональность смены схем из настроек IntelliJ IDEA — делает такую смену настроек более быстрой.

3.2.4.16. *Type Hierarchy (Иерархия типов)*

"Горячие" клавиши: <Ctrl>+<H>.

Данная команда открывает инструментальное окно **Hierarchy** (Иерархия) (*см. раздел 3.4.8.9*) и отображает в нем иерархию класса, для которого была вызвана (рис. 3.42).

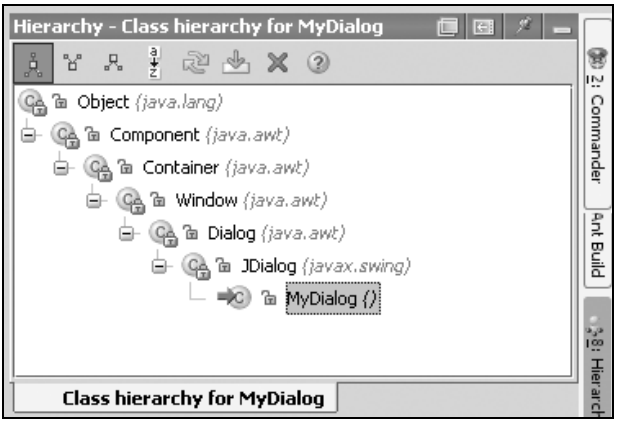


Рис. 3.42. Результат работы команды **Type Hierarchy**

3.2.4.17. *Method Hierarchy (Иерархия методов)*

"Горячие" клавиши: <Ctrl>+<Shift>+<H>.

Команда работает аналогично команде **Type Hierarchy** (Иерархия типов) (*см. раздел 3.2.4.16*), за тем исключением, что отображает иерархию для выбранного метода, а не типа (рис. 3.43).

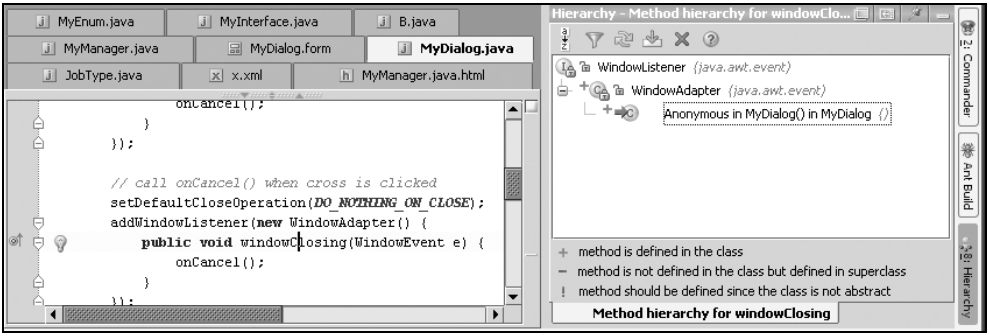


Рис. 3.43. Результат работы команды **Method Hierarchy**

3.2.4.18. *Call Hierarchy (Иерархия вызовов)*

"Горячие" клавиши: <Ctrl>+<Alt>+<H>.

Команда отображает иерархию вызовов метода и работает аналогично двум другим иерархическим командам (*см. разделы 3.2.4.16, 3.2.4.17*), (рис. 3.44).

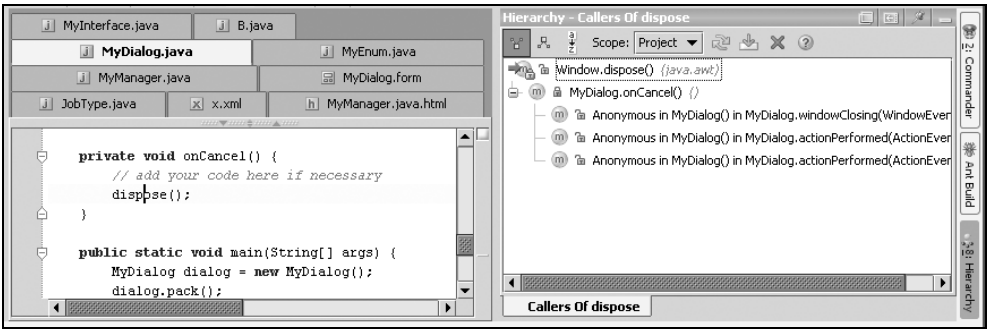


Рис. 3.44. Результат работы команды **Call Hierarchy**

3.2.4.19. *Folding (Сокращенный/Развернутый вид)* [1]

Группа команд, управляющих отображением содержимого файла в сокращенном или развернутом виде. Прежде, чем прочитать об этих командах, посмотрите на рис. 3.45. В левой части окна редактора (между левым полем и самим окном редактирования) размещаются парные метки, обращенные друг к другу углом. Если подвести к одной из таких меток указатель мыши, между парными метками появится тонкая черная линия, их соединяющая — это обозначение границ свертывания. Если на одну из меток щелкнуть мышью, текст, расположенный в пределах меток, свернется, а сами метки превратятся в одну квадратную со значком + (рис. 3.45). При этом свернутый текст будет отображаться как две фигурные скобки с многоточием между ними {}.

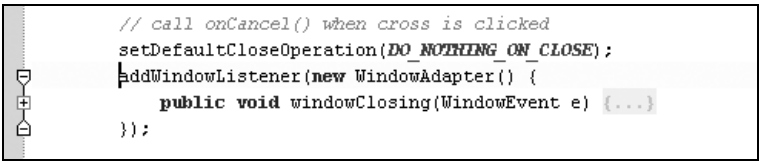


Рис. 3.45. Результат сворачивания

Всего в группе **Folding** (Сокращенный/Развернутый вид) восемь команд, первые шесть из которых парные:

- **Expand/Collapse** (Развернуть/Свернуть) — позволяет свернуть/развернуть развернутый/свернутый текст по месту расположения курсора;
- **Expand All/Collapse All** (Развернуть все/Свернуть все) — свернуть/развернуть все развернутые/свернутые части текста в файле;

[1] Приведенный перевод английского слова "Folding" на русский язык несколько кривой, сделать однословный перевод достаточно сложно (от англ. Folding — складывать).

- **Expand Javadocs/Collapse Javadocs** (Развернуть Javadoc/Свернуть Javadoc) — свернуть/развернуть JavaDoc к текущему коду.

Последние две команды:

- **Fold Selection** (Свернуть выделение) <Ctrl>+<.> — свернуть/развернуть выделенный текст;
- **Fold Code Block** (Свернуть блок кода) <Ctrl>+<Shift>+<.> — свернуть/развернуть блок кода.

3.2.5. Меню *Go To (Перейти к)*

Команды из этого пункта меню предназначены для навигации по проекту и внутри кода.

3.2.5.1. *Class (Класс)*

"Горячие" клавиши: <Ctrl>+<N>.

Перейти к указанному классу (рис. 3.46).

Если в проекте имеются классы, имя которых подходит под вводимое имя, они будут отображены в списке. Для данной команды есть возможность производить поиск класса для перехода не только внутри проекта, но также и в прочих классах (Include non-project classes), например, среди классов из библиотек.

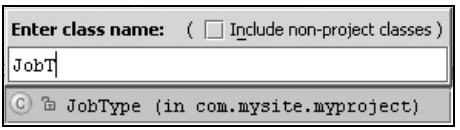


Рис. 3.46. Варианты классов для перехода

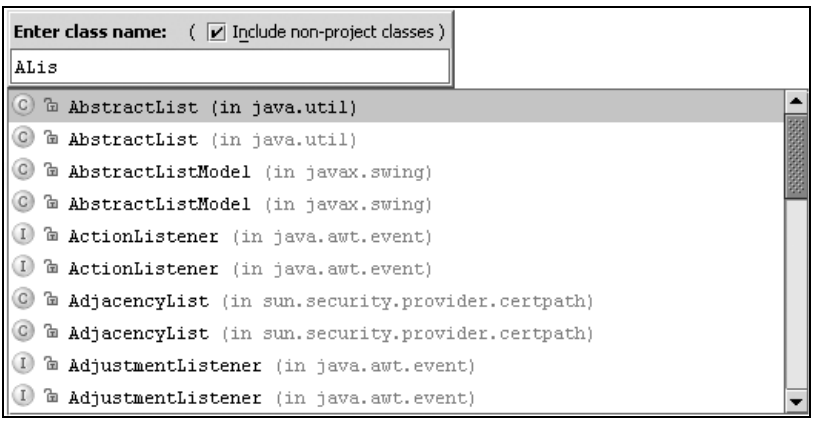


Рис. 3.47. Варианты классов для перехода найденных по маске из заглавных букв

Одной из замечательных возможностей данной команды является то, что в имени класса можно использовать маску *, а также писать имя класса в неполном виде — вводя только заглавные буквы составных слов в имени класса (рис. 3.47).

3.2.5.2. *File (Файл)*

"Горячие" клавиши: <Ctrl>+<Shift>+<N>.

Перейти к указанному файлу. Данная команда работает аналогично команде **Class** (Класс) (*см. раздел 3.2.5.1*), только ищет не классы, а файлы в проекте (рис. 3.48).

Рис. 3.48. Список файлов для перехода

В качестве дополнительной опции поиска (включенной по умолчанию), имеется возможность включать в поиск и Java-файлы.

3.2.5.3. *Symbol (Символ)*

"Горячие" клавиши: <Ctrl>+<Alt>+<Shift>+<N>.

Перейти к указанному символу. Под символом понимаются имена методов и поля классов (рис. 3.49).

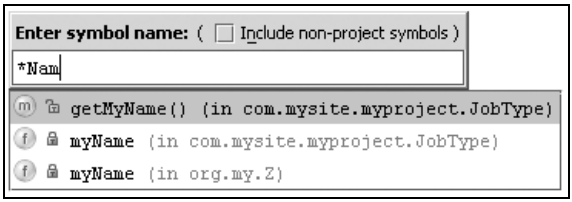


Рис. 3.49. Список символов для перехода

Функциональность поиска данной команды аналогична командам **Class** (Класс) и **File** (Файл) (*см. разделы 3.2.5.1, 3.2.5.2*).

3.2.5.4. *Line (Строка)*

"Горячие" клавиши: <Ctrl>+<G>.

Команда перехода к указанной строке (рис. 3.50).

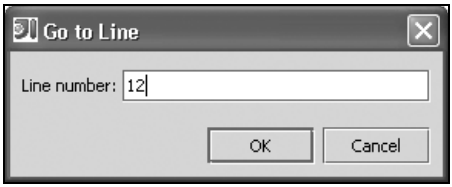


Рис. 3.50. Диалог перехода к указанной строке в файле

3.2.5.5. *Declaration (Декларация)*

"Горячие" клавиши: <Ctrl>+<B>.

Команда перехода к декларации (определению) какого-либо символа. Данная команда активна, когда курсор находится поверх символа, который определен где-то в другом месте кода. Например, это может быть имя класса, метода, поля класса или имя переменной. При выполнении этой команды курсор автоматически перемещается к месту в коде, где находится декларация указанного символа. Если символ определен в другом файле, этот файл будет открыт в редакторе.

Данную команду можно также выполнять с помощью нажатия левой кнопки мыши с одновременно нажатой клавишей <Ctrl>.

3.2.5.6. *Implementation(s) (Имплементации)*

"Горячие" клавиши: <Ctrl>+<Alt>+<B>.

Команда перехода к имплементации указанного символа (класса, интерфейса или метода). Данная команда, по сути, является обратной к команде **Declaration** (Определение) (*см. раздел 3.2.5.5*). Если обнаружено несколько имплементаций, будет выведен список со всеми найденными (рис. 3.51).

Данная команда находит имплементации не только в файлах проекта, но и в классах библиотек.

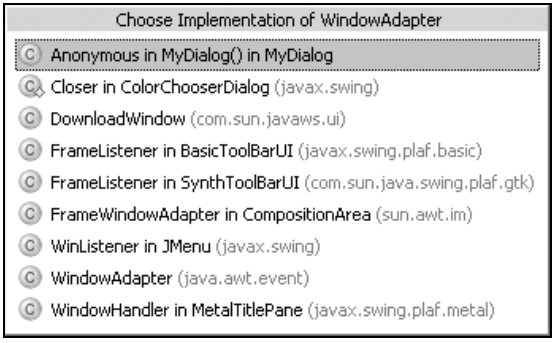


Рис. 3.51. Результат работы команды **Go To | Implementation(s)**

3.2.5.7. *Called Implementation(s) (Вызванные имплементации)*

"Горячие" клавиши: <Ctrl>+<Alt>+<Shift>+<B>.

Команда работает аналогично команде **Implementations** (Имплементации) (*см. раздел 3.2.5.6*), но находит только те имплементации, которые были вызваны где-то в коде.

Примечание

К сожалению, в IntelliJ IDEA 4.5 есть ошибка в работе этой команды, из-за которой команда работает полностью идентично команде **Implementations** (Имплементации). Данная ошибка уже исправлена, и команда будет работать правильно в следующей версии IntelliJ IDEA.

3.2.5.8. *Type Declaration (Определение типа)*

"Горячие" клавиши: <Ctrl>+<Shift>+<B>.

Команда перехода к определению типа переменной или поля класса. Команда активна, только если курсор находится на имени переменной или поля класса.

3.2.5.9. *Super Method (Метод из суперкласса)*

"Горячие" клавиши: <Ctrl>+<U>.

Команда перехода к определению метода из суперкласса. Команда активна, только если курсор находится на имени метода, который переопределяет (override) метод в суперклассе, либо в интерфейсе.

3.2.5.10. *Next/Previous Highlighted Error (Следующая/Предыдущая подсвеченная ошибка)*

"Горячие" клавиши: <F2>/<Shift>+<F2>.

IntelliJ IDEA может находить ошибки в коде и подсвечивать их красной волнистой линией. Две данных команды позволяют переходить между этими ошибками в коде.

3.2.5.11. *Next/Previous Method (Следующий/Предыдущий метод)*

"Горячие" клавиши: <Alt>+<Down>/<Alt>+<Up>.

Команды перехода к следующему/предыдущему методу в классе или интерфейсе относительно текущей позиции курсора в редакторе.

3.2.5.12. *Back/Forward (Назад/Вперед)*

"Горячие" клавиши: <Ctrl>+<Alt>+<Left>/<Ctrl>+<Alt>+<Right>.

Команды переводят курсор к предыдущему/следующему месту в редакторе IntelliJ IDEA.

IntelliJ IDEA запоминает места в редактируемых файлах и позволяет переходить между ними с помощью этих команд.

3.2.5.13. *Last Edit Location (Место последнего редактирования)*

"Горячие" клавиши: <Ctrl>+<Shift>+<Backspace>.

Команда перехода к месту последнего редактирования.

3.2.5.14. *Go To Next/Prev Change (Перейти к следующему/предыдущему изменению)*

Данные команды становятся доступными, если проект использует какую-либо систему контроля версий (*см. главу 12*). С помощью этих команд можно переходить между локальными изменениями в файлах. Подробнее о работе с этими командами написано в *разделе 12.6*.

3.2.6. Меню *Code (Код)*

В разделе меню **Code** (Код) собраны команды, упрощающие создание и модификацию кода.

3.2.6.1. *Override Methods (Переопределить методы)*

"Горячие" клавиши: <Ctrl>+<O>.

Команда позволяет автоматически генерировать код для переопределяемых (override) методов класса. Окно, в котором происходит выбор методов класса для переопределения, показано на рис. 3.52.

В открывшемся модальном диалоге можно выбрать методы наследуемых классов, которые должны быть переопределены в текущем классе.

У данного диалога есть собственная панель инструментов, в которой есть четыре кнопки:

- **Sort Alphabetically** (Сортировать по алфавиту) (<Alt>+<A>) — включает/выключает сортировку названий методов по алфавиту;
- **Show Classes** (Показывать классы) (<Alt>+<C>) — включает/выключает группировку методов по классам. На рис. 3.52 такая группировка включена;
- **Expand All** (Развернуть все) (<Alt>+<=>) — развернуть все свернутые группы методов;
- **Collapse All** (Свернуть все) (<Alt>+<->) — свернуть все развернутые группы методов.

Также в данном диалоге доступны две дополнительные опции:

- **Insert @Override** (Вставлять аннотацию `@override`) — данная опция добавит в описания переопределяемых методов аннотацию `@override`. Эта опция

доступна, если в проекте включена поддержка возможностей языка Java 5 (*см. главу 11*);

- **Copy JavaDoc** (Копировать JavaDoc) — при включении данной опции будут скопированы описания JavaDoc из классов, методы которых переопределяются.

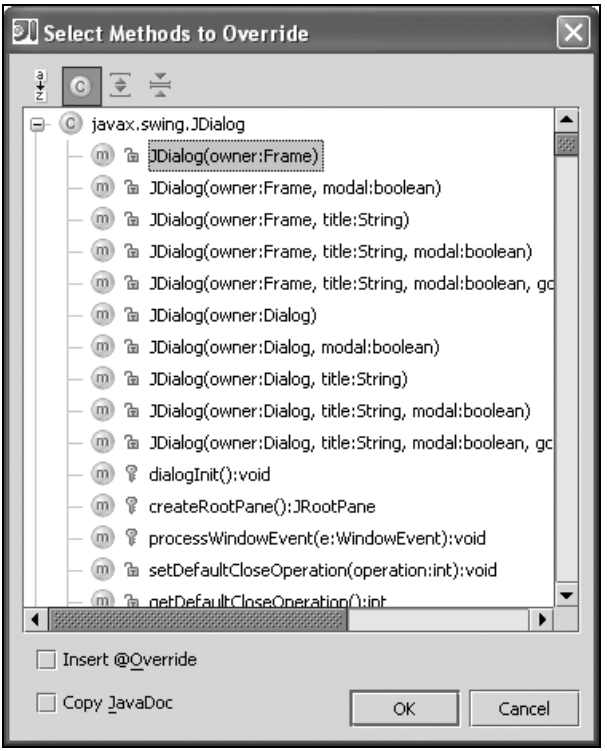


Рис. 3.52. Результат работы команды **Override Methods**

3.2.6.2. *Implement Methods (Имплементировать методы)*

"Горячие" клавиши: <Ctrl>+<I>.

Команда позволяет автоматически генерировать код для имплементируемых методов интерфейса или абстрактного класса. Выбор методов из наследуемых интерфейсов или абстрактных классов для имплементации показан на рис. 3.53.

На рис. 3.53 для имплементации доступен только один метод `actionPerformed(e:ActionEvent)`, определенный в интерфейсе `ActionListener`. Из дополнительных опций доступна только опция копирования JavaDoc.

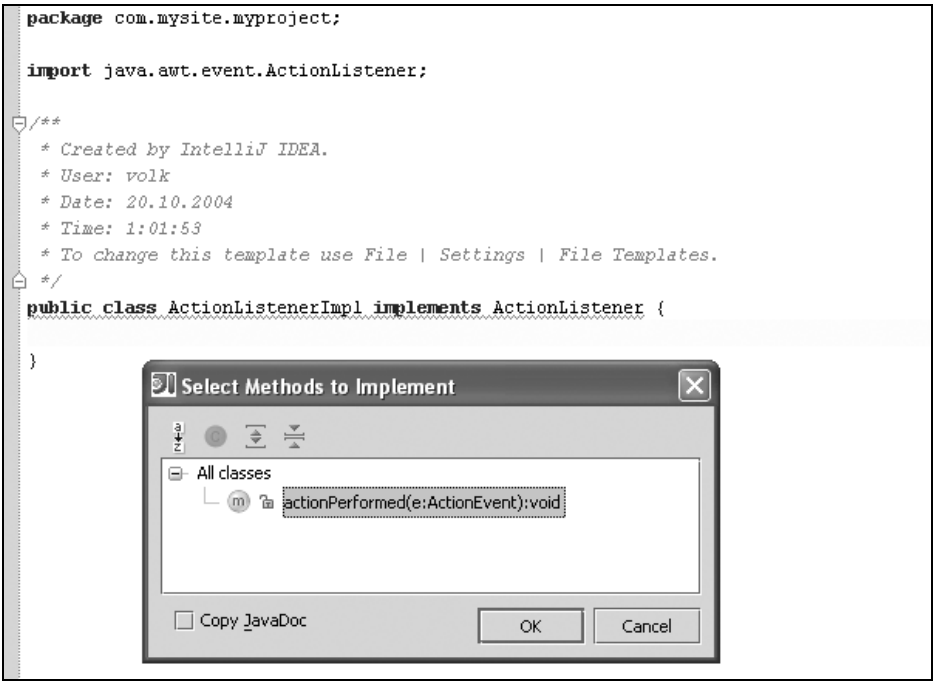


Рис. 3.53. Результат работы команды **Implement Methods**: выбор метода для имплементации

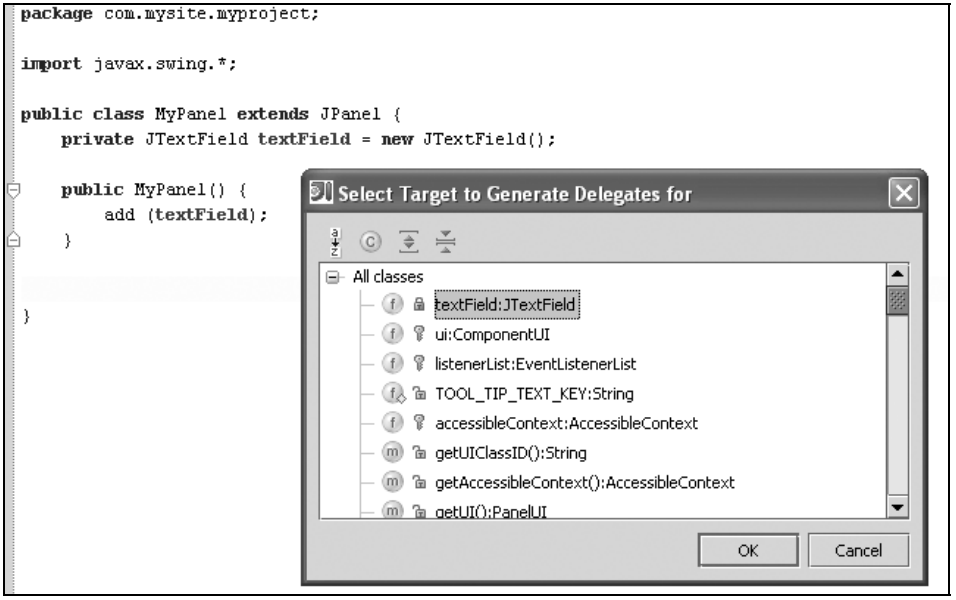


Рис. 3.54. Выбор поля класса для делегирования метода

3.2.6.3. *Delegate Methods (Делегировать методы)*

Данная команда позволяет создать делегированные методы для полей класса (рис. 3.54).

На рис. 3.54 мы видим класс, наследующийся от класса `JPanel` панели в `Swing`. В нашей панели есть текстовое поле `textField`. Для него мы создаем метод `getText()`, который будем делегировать (рис. 3.55).

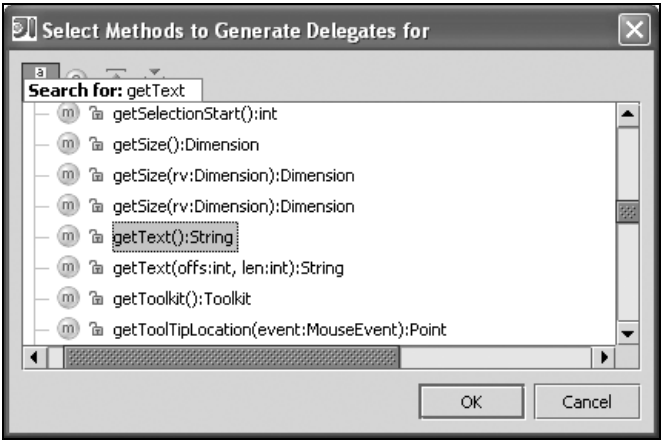


Рис. 3.55. Выбор метода для делегирования

Результат делегирования метода представлен в листинге 3.3 — был создан метод `getText()` в классе `MyPanel`, который делегирует метод `getText()` поля `textField`.

Листинг 3.3

```
package com.mysite.myproject;

import javax.swing.*;

public class MyPanel extends JPanel {
    private JTextField textField = new JTextField();

    public MyPanel() {
        add (textField);
    }
```

```
    public String getText() {
        return textField.getText();
    }
}
```

Данную команду очень удобно использовать при создании оберток (wrapper) и скелетов (stub) для кода.

3.2.6.4. *Surround With (Завернуть в...)*

"Горячие" клавиши: <Ctrl>+<Alt>+<T>.

Данная команда позволяет "завернуть" или надстроить выбранный фрагмент кода несколькими вариантами "обертки" или надстройки (рис. 3.56).

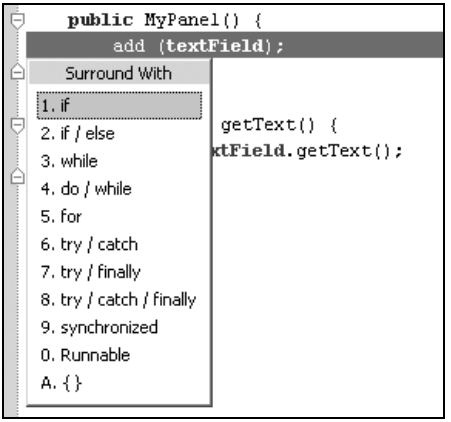


Рис. 3.56. Варианты "обертки" для команды **Surround With**

Всего есть одиннадцать вариантов "обертки" выбранного фрагмента:

1. **if** — условный оператор `if`.
2. **if / else** — условный оператор `if...else` — если то.
3. **while** — оператор цикла `while`.
4. **do / while** — оператор цикла `do...while`.
5. **for** — оператор цикла `for`.
6. **try / catch** — оператор обработки исключения `try...catch`.
7. **try / finally** — оператор финализации (finalization).
8. **try / catch / finally** — оператор обработки исключения и финализации (finalization).
9. **synchronized** — оператор синхронизации.

10. **Runnable** — блок кода, заключенный внутри метода `run()` имплементации интерфейса `Runnable`.
11. **{}** — фигурные скобки.

В табл. 3.1 приведены примеры работы всех вариантов "обертки" для выделенного блока с рис. 3.56.

Таблица 3.1. *Примеры работы команды* ***Surround With*** *(Завернуть в...)*

Вариант "обертки"	Результат работы команды

If	`if () {`
	`  add (textField);`
	`}`

if / else	`if () {`
	`  add (textField);`
	`} else {`
	`}`

while	`while (true) {`
	`  add (textField);`
	`}`

do / while	`do {`
	`  add (textField);`
	`} while (true);`

for	`for () {`
	`  add (textField);`
	`}`

try / catch	`try {`
	`  add (textField);`
	`} catch (Exception e) {`
	`  e.printStackTrace();`
	`}`

try / finally	`try {`
	`  add (textField);`
	`} finally {`
	`}`

try / catch / finally	`try {`
	`  add (textField);`
	`} catch (Exception e) {`
	`  e.printStackTrace();`
	`} finally {`
	`}`

Таблица 3.1 *(окончание)*

Вариант "обертки"	Результат работы команды

synchronized	`synchronized () {`
	`  add (textField);`
	`}`

Runnable	`Runnable runnable = new Runnable() {`
	`  public void run() {`
	`  add (textField);`
	`  }`
	`};`

{}	`{`
	`  add (textField);`
	`}`

Данная команда очень помогает сэкономить время при написании кода.

3.2.6.5. *Generate (Сгенерировать)*

"Горячие" клавиши: <Alt>+<Insert>.

Эта команда похожа на команду **New** (Создать) из меню **Edit** (Редактирование) (*см. раздел 3.2.2.10*), она даже имеет то же самое сочетание "горячих" клавиш — <Alt>+<Insert>. Это сделано неспроста. Дело в том, что по своей сути эти команды идентичны — обе выполняют создание новых объектов, только в случае команды **New** (Создать) создаются файлы, классы или папки, а команда **Generate** (Сгенерировать) создает различные элементы кода (рис. 3.57).

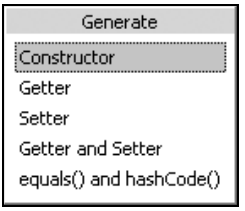


Рис. 3.57. Список доступных вариантов для генерации

Всего есть пять вариантов того, что может быть сгенерировано:

- **Constructor** (Конструктор) — конструктор класса;
- **Getter** (Метод для получения значения поля класса);
- **Setter** (Метод для установки значения поля класса);

- **Getter and Setter** (Методы для получения и установки значения поля класса);
- **equals () and hashCode()** — методы `equals()` и `hashCode()` для класса.

Рассмотрим на примере работу этой команды на классе, приведенном в листинге 3.4.

Листинг 3.4

```
public class Circle {
    private int radius;
}
```

В случае создания конструктора (Constructor), IntelliJ IDEA попросит выбрать поля класса, которые должны быть проинициализированы этим конструктором (рис. 3.58). При создании методов для установки и получения значения (Getter, Setter), а также для создания методов `equals()` и `hashCode()`, IntelliJ IDEA попросит вас указать поля, для которых вы хотите сгенерировать указанные методы. Если для класса из листинга 3.4 сгенерировать все возможные методы, мы получим код, приведенный в листинге 3.5.

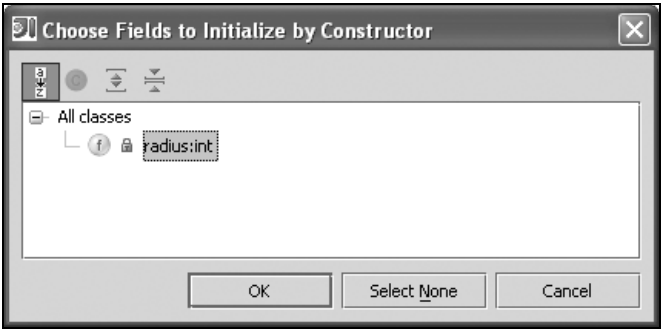


Рис. 3.58. Выбор полей класса для инициализации конструктором

Листинг 3.5

```
public class Circle {
    private int radius;
    // Generate | Constructor
    public Circle(int radius) {
        this.radius = radius;
```

```
    }
    // Generate | Getter
    public int getRadius() {
        return radius;
    }
    // Generate | Setter
    public void setRadius(int radius) {
        this.radius = radius;
    }
    // Generate | equals() and hashCode()
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Circle)) return false;

        final Circle circle = (Circle) o;

        if (radius != circle.radius) return false;

        return true;
    }

    public int hashCode() {
        return radius;
    }
}
```

Данная команда также существенно ускоряет написание кода.

3.2.6.6. *Complete Code (Завершить код)*

Данный подраздел меню объединяет три команды завершения кода:

- **Basic** (Простой) (<Ctrl>+<Space>) — простой метод завершения кода — делает подстановку названий классов, полей класса, методов, переменных в пределах области видимости, а также ключевых слов в Java. Данный метод завершения использует простой строковый анализ текста, который вы хотите завершить;
- **SmartType** (Умный тип) (<Ctrl>+<Shift>+<Space>) — умное завершения типа — данный метод завершения, в отличие от простейшего, производит дополнительный анализ контекста;

- **Class Name** (Имя класса) (<Ctrl>+<Alt>+<Space>) — умное завершение имени класса — метод позволяет завершить название класса, даже если он не доступен в текущей области видимости. При этом класс, имя которого будет завершено, автоматически добавится в список импортов.

Методы завершения кода значительно упрощают написание качественного кода. Они будут подробно рассмотрены в *разделе 6.1.5*.

3.2.6.7. *Insert Live Template (Вставить "живой шаблон")*

"Горячие" клавиши: <Ctrl>+<J>.

"Живые шаблоны" — еще одна из интересных возможностей IntelliJ IDEA — это специальное сочетание символов, которые могут "раскрываться" в некоторый блок кода (рис. 3.59).

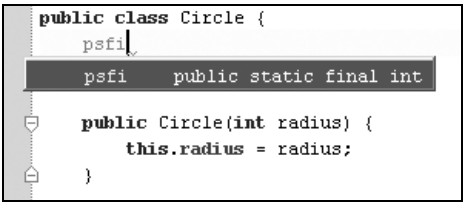


Рис. 3.59. Ввод буквенного сочетания и запуск "живого шаблона"

Набираем в тексте специальное буквенное сочетание `psfi`, а затем запускаем команду **Insert Live Template** (Вставить "живой шаблон"). В результате IntelliJ IDEA отобразит список возможных "живых шаблонов" (в данном случае — один шаблон), вставив который мы получим код, приведенный на рис. 3.60.

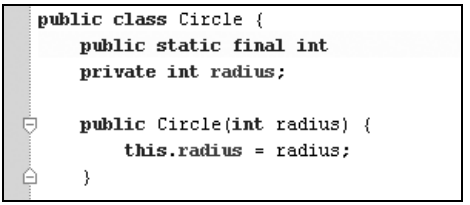


Рис. 3.60. Результат работы "живого шаблона" `psfi`

Сокращение `psfi` превратилось в `public static final int`.

В IntelliJ IDEA есть очень много встроенных "живых шаблонов". Некоторые из них, подобно описанному выше, работают как простые сокращения, другие — более интеллектуальны и могут подстраиваться под текущий контекст. Их использование существенно ускоряет написание программного кода. Также есть возможность создавать собственные "живые шаблоны".

Возможности "живых шаблонов" будут подробно рассмотрены в *разделах 4.1.13, 6.3.2.*

3.2.6.8. *Surround with Live Template (Обернуть "живым шаблоном")*

"Горячие" клавиши: <Ctrl>+<Alt>+<J>.

Данная команда является своеобразной производной двух других команд: **Insert Live Template** (Вставить "живой шаблон") и **Surround With** (Завернуть в...) (*см. разделы 3.2.6.4, 3.2.6.7*).

Есть три встроенных "живых шаблона", которые могут работать с этой командой:

- **Surround with {}** — обертывает указанный блок фигурными скобками;
- **Surround with ()** — обертывает указанный блок круглыми скобками;
- **Surround with <tag></tag>** — обертывает тегом. Данный "живой шаблон" работает только для XML-кода.

Подробнее о "живых шаблонах" можно прочитать в *разделах 4.1.13, 6.3.2.*

3.2.6.9. *Comment with Line Comment (Закомментировать построчно)*

"Горячие" клавиши: <Ctrl>+</>.

Данная команда помещает указанный блок текста в построчный комментарий. Если применить ее к одному и тому же блоку текста повторно, комментирование будет снято. В случае, если редактируемый код не поддерживает строчных комментариев (например XML-код), будет вставлен блочный комментарий.

3.2.6.10. *Comment with Block Comment (Закомментировать блочным комментарием)*

"Горячие" клавиши: <Ctrl>+<Shift>+</>.

Команда помещает указанный блок текста в блочный комментарий. Повторное применение команды для того же самого блока текста снимает комментирование.

3.2.6.11. *Auto-Indent Lines (Автоматический отступ красной строки)*

"Горячие" клавиши: <Ctrl>+<Alt>+<I>.

Команда автоматически устанавливает отступ красной строки (отступ слева) для указанного блока текста, используя настройки стиля кода (*см. раздел 4.1.7*).

3.2.7. Меню *Analyze (Анализ)*

Пункт меню **Analyze** (Анализ) содержит команды анализа кода. Под анализом в данном контексте понимается исследование широкого круга особенностей построения кода, которые могут быть улучшены в том или ином качестве. IntelliJ IDEA обладает очень широкими возможностями анализа кода.

3.2.7.1. *Inspect Code (Инспекция кода)*

Данная команда запускает набор так называемых инспекций (inspections) — функций проверки кода. Существует около 400 таких функций, каждая из которых обладает собственными настройками и параметрами. Результат работы инспекции носит рекомендательный характер. Например, одна из функций проверки занимается поиском неиспользуемых переменных. И если такие переменные будут найдены, прямо из отчета работы инспекции эти переменные можно будет удалить.

Детальный обзор функций инспекции кода и их настроек можно найти в *разделе 4.1.5*, а об их использовании можно прочитать в *разделе 6.4.4*.

3.2.7.2. *View Offline Inspection Results (Просмотр результатов инспекции кода)*

В комплекте поставки IntelliJ IDEA содержится отдельная утилита для проведения инспекции кода. Дело в том, что инспекции могут занимать достаточно много времени, особенно если обработке подвергается очень большой проект. Поэтому запускать инспекции непосредственно из-под IntelliJ IDEA бывает нецелесообразно. Для этого и создана отдельная программа-инспектор. Результат работы этой программы заносится в специальный отчет, который затем можно просмотреть непосредственно в IntelliJ IDEA с помощью этой команды. Подробнее об этом можно прочитать в *разделе 6.4.4*.

3.2.7.3. *Analyze Dependencies (Анализ зависимостей)*

Данная команда производит поиск зависимостей между отдельными частями проекта — модулями, пакетами и классами. Эта команда бывает полезной в случае, если вы собираетесь выделить некоторый набор классов в отдельный модуль или библиотеку. Подробнее об анализе зависимостей можно прочитать в *разделе 6.4.2*.

3.2.7.4. *Locate Duplicates (Поиск повторений)*

Данная команда производит поиск повторений в коде. Очень часто, особенно в больших проектах, несмотря на различные объектно-ориентированные технологии, код пишется с помощью простой операции Copy-Paste (скопи-

ровал-вставил). В результате чего программа начинает разрастаться повторяющимися кусками кода, многие из которых затем отдельно модифицируются. В итоге получается полная каша. Как раз для того, чтобы этого избежать, и существует данная команда.

Процедура анализа текста с помощью возможностей IntelliJ IDEA подробно рассмотрена в *разделе 6.4.3*.

3.2.8. Меню *Refactor (Рефакторинг)*

Данный пункт меню содержит полный перечень команд для проведения различных рефакторингов. Всего имеется 25 базовых команд. В данном разделе мы приведем лишь краткие описания этих команд. Подробнее рефакторинг с помощью IntelliJ IDEA рассмотрен в *главе 8*.

Многие команды рефакторинга работают не только на Java-коде, но и на обычных файлах.

- **Rename** (Переименовать) (<Shift>+<F6>). Рефакторинг переименования — все вхождения данного символа заменяются его новым названием (*см. раздел 8.1*).
- **Change Signature** (Изменить сигнатуру) (<Ctrl>+<F6>). Изменение сигнатуры метода, т. е. его названия, возвращаемого значения и входных параметров (*см. раздел 8.2*).
- **Make Method Static** (Сделать метод статичным). Метод (экземпляра) класса становится статическим (*см. раздел 8.3*).
- **Convert To Instance Method** (Конвертировать в обычный метод). Обратная команда, которая статический метод преобразует в метод экземпляра класса (*см. раздел 8.4*).
- **Move** (Переместить) (<F6>). Перемещает произвольные сущности, например классы, в другие пакеты, файлы, каталоги и т. п. (*см. раздел 8.5*).
- **Copy** (Скопировать) (<F5>). Копирует произвольные сущности (*см. раздел 8.6*).
- **Safe Delete** (Безопасное удаление) (<Alt>+<Delete>). Данная команда позволяет удалить некоторую сущность, предварительно проверив, не повредит ли такое удаление остальному коду (*см. раздел 8.7*).
- **Extract Method** (Выделить метод) (<Ctrl>+<Alt>+<M>). Команда выделяет метод из кода (*см. раздел 8.8*).
- **Replace Method Code Duplicates** (Заменить методом повторяющийся код). Данная команда позволяет заменить все вхождения повторяющегося кода вызовом метода, который будет автоматически создан из этого кода (*см. раздел 8.9*).

- **Introduce Variable** (Ввести переменную) (<Ctrl>+<Alt>+<V>). Команда позволяет преобразовать некоторое выражение в локальную переменную (*см. раздел 8.10*).
- **Introduce Field** (Ввести поле класса) (<Ctrl>+<Alt>+<F>). Команда действует аналогично предыдущей, но создает поле класса (*см. раздел 8.11*).
- **Introduce Constant** (Ввести константу) (<Ctrl>+<Alt>+<C>). Команда позволяет преобразовать некоторое постоянное значение, встречающееся в коде, в именованную константу (*см. раздел 8.12*).
- **Introduce Parameter** (Ввести параметр) (<Ctrl>+<Alt>+<P>). Команда заменяет вхождения некоторого выражения на новый параметр метода (*см. раздел 8.13*).
- **Extract Interface** (Выделить интерфейс). Команда выделяет интерфейс из одного или более методов класса (*см. раздел 8.14*).
- **Extract Superclass** (Выделить базовый класс). Команда выделяет базовый класс из полей и методов класса (*см. раздел 8.15*).
- **Use Interface Where Possible** (Использовать интерфейс, где это возможно). Команда заменяет места в коде, в которых используется класс, на один из его базовых классов или интерфейс, если это возможно (*см. раздел 8.16*).
- **Pull Members Up** (Перемещение членов класса вверх). Команда позволяет переместить члены класса (методы, поля) в его базовый класс (*см. раздел 8.17*).
- **Push Members Down** (Перемещение членов класса вниз). Обратная команда, которая переносит члены базового класса в один из его производных классов (*см. раздел 8.18*).
- **Replace Inheritance with Delegation** (Заменить наследование делегированием). Данная команда позволяет заменить наследование некоторых методов класса делегированием (*см. раздел 8.19*).
- **Inline** (Подстановка в код) (<Ctrl>+<Alt>+<N>). Данная команда заменяет вызов метода или статического поля класса на его тело (*см. раздел 8.20*).
- **Convert Anonymous to Inner** (Конвертировать анонимный класс во внутренний). Команда позволяет конвертировать анонимный класс во внутренний класс (*см. раздел 8.21*).
- **Encapsulate Fields** (Инкапсулировать поля). Данная команда позволяет заменить прямое использование полей класса на вызовы соответствующих методов получения и установки значения (*см. раздел 8.22*).
- **Replace Temp with Query** (Заменить временное использование вызовом). Команда заменяет использование локальной переменной, значение которой не меняется после инициализации, на вызов метода, возвращающего ее значение (*см. раздел 8.23*).

- **Replace Constructor with Factory Method** (Заменить конструктор методом фабрики). Команда заменяет конструктор на метод фабрики. Конструктор при этом получает спецификацию доступа `private`, а в класс добавляется новый статический метод, вызывающий этот конструктор (*см. раздел 8.24*).
- **Generify** ("Генерификация" — параметризация классов). Данная команда позволяет заменить старую нотацию языка Java на новую, использующую параметризованные классы (generics), появившиеся в пятой версии языка Java (Java 5) (*см. раздел 8.25*). Подробнее о работе с параметризованными классами в IntelliJ IDEA можно прочитать в *главе 11*.

3.2.9. Меню *Build (Сборка)*

Данный пункт меню содержит команды, предназначенные для компиляции и сборки проекта.

3.2.9.1. *Rebuild Project (Пересобрать проект)*

Данная команда производит полную компиляцию проекта, включая компиляцию всех его составляющих модулей.

Так как компиляция — довольно длительный процесс, а уж компиляция всего проекта особенно, IntelliJ IDEA отображает окно с индикатором хода компиляции. В окне отображается количество файлов, которые IntelliJ IDEA уже успела просмотреть, а также количество классов, которые были из них скомпилированы (рис. 3.61).

Рис. 3.61. Информация о ходе компиляции

Есть возможность переключить компиляцию в фоновый (background) режим, при этом продолжать работать с IntelliJ IDEA, а компиляция будет продолжаться. Для переключения в фоновый режим нужно нажать кнопку **Background** (Фон).

Если в ходе компиляции были обнаружены какие-либо ошибки или замечания, они будут выведены в инструментальном окне **Messages** (Сообщения) (*см. раздел 3.4.8.1*), (рис. 3.62).

В данном случае компилятор обнаружил использование одного устаревшего метода в классе `MyDialog`.

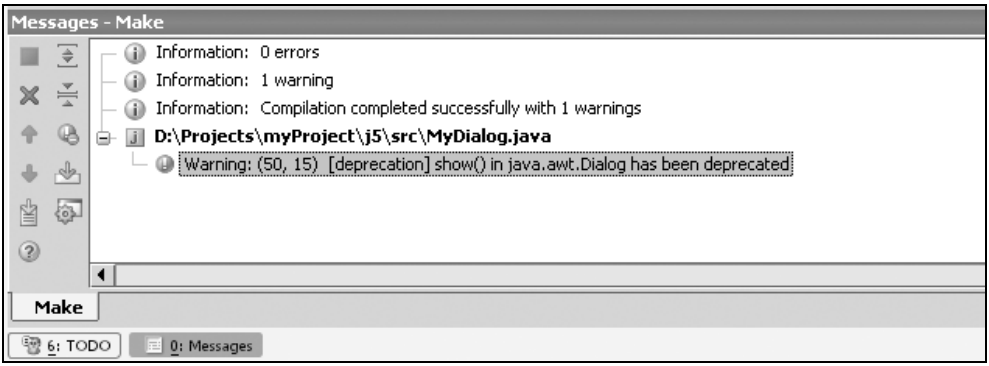


Рис. 3.62. Инструментальное окно **Messages** с информацией о результатах компиляции

3.2.9.2. *Make Project (Собрать проект)*

"Горячие" клавиши: <Ctrl>+<F9>.

Данная команда компилирует только те файлы проекта, которые изменились с момента последней компиляции (в том числе и вновь созданные файлы).

3.2.9.3. *Make Module (Собрать модуль)*

Команда выполняет компиляцию всех классов из выбранного модуля, которые изменились с момента последней компиляции.

3.2.9.4. *Compile (Компилировать)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F9>.

Количество классов, которые компилирует данная команда, зависит от контекста ее вызова. Она может быть вызвана для какого-либо конкретного класса, пакета, модуля, всего проекта. Команда компилирует все попавшие к ней классы.

3.2.9.5. *Generate Ant Build (Сгенерировать файл для сборки утилитой Ant)*

Данная команда создает файл для сборки проекта в формате утилиты Ant. Выполнив эту команду, вы увидите диалог, представленный на рис. 3.63.

В диалоге есть следующие опции:

- **Generate multiple-file ant build** — сгенерировать несколько файлов для сборки. Если выбрать эту опцию, для каждого модуля в проекте будет создан отдельный файл. Чтобы выполнить сборку указанными файлами, требуется Ant версии 1.6 или выше;
- **Generate single-file ant build** — сгенерировать один файл.

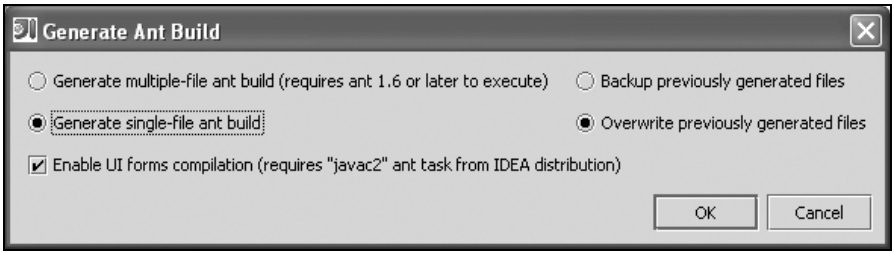


Рис. 3.63. Генерация файла в формате Ant для сборки проекта

Также имеется возможность сделать резервную копию предыдущих файлов, если они были (**Backup previously generated files**), или перезаписать эти файлы (**Overwrite previously generated files**).

Кроме указанных опций есть еще одна: **"Enable UI forms compilation"** (Разрешить компиляцию форм интерфейса пользователя). В IntelliJ IDEA есть встроенный визуальный редактор интерфейса пользователя (*см. главу 7*), который работает с формами в специальном формате (XML) — javac2. Данная опция позволяет включить в создаваемый файл для сборки Ant-команды для компиляции этих форм, для чего будет использована специальная команда `javac2`, входящая в поставку IntelliJ IDEA. Если эта опция выключена, будет использована стандартная команда компиляции `javac`.

Результатом работы данной команды будет XML-файл (или несколько файлов, если была выбрана соответствующая опция) для сборки проекта утилитой Ant, расположенный в директории с файлом проекта.

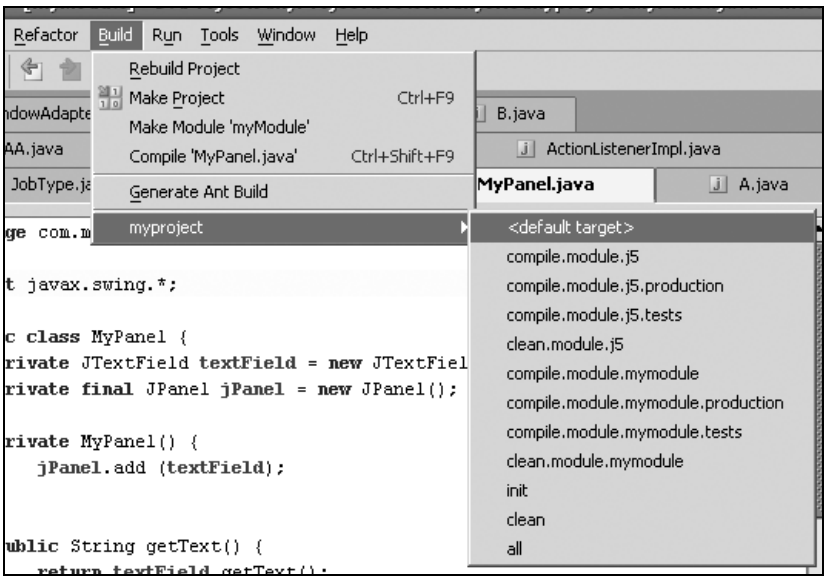


Рис. 3.64. Дополнительные команды в меню **Build**

Если проект, открытый в IntelliJ IDEA, имеет сконфигурированные файлы для сборки утилитой Ant, меню **Build** (Сборка) будет дополнено командами из этих файлов (рис. 3.64).

Подробнее об утилите Ant и ее поддержке в IntelliJ IDEA написано в *главе 13*.

3.2.10. Меню *Run (Запуск)*

Данный пункт меню содержит команды для запуска и отладки приложений. Более подробно запуск и отладка приложений рассмотрены в *главе 9*.

Для запуска и отладки приложений IntelliJ IDEA применяет так называемые конфигурации запуска/отладки — набор настроек, используемых при старте приложений. Чтобы дальнейшее описание команд из меню **Run** (Запуск) было более понятным, мы начнем их описание с команды управления этими конфигурациями. Она стоит третьей по счету в меню **Run** (Запуск).

3.2.10.1. *Edit Configurations (Редактировать конфигурации)*

Данная команда открывает диалог настройки конфигураций (рис. 3.65).

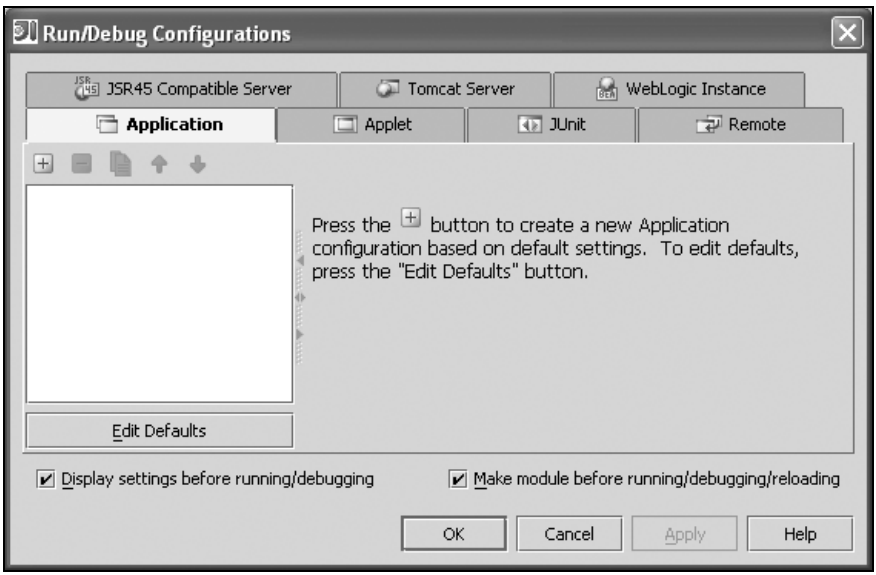


Рис. 3.65. Диалог конфигураций запуска/отладки

В этом диалоге есть набор закладок, каждая из которых позволяет настроить конфигурацию конкретного типа:

- **Application** — конфигурации для запуска Java-приложений;
- **Applet** — конфигурации для запуска Java-апплетов;

- **JUnit** — конфигурации для запуска тестов с помощью утилиты JUnit;
- **Remote** — конфигурации для удаленной отладки;
- **JSR45 Compatible Server** — конфигурации для запуска и отладки Web-серверов и серверов приложений, совместимых с JSR45;
- **Tomcat Server** — конфигурации для запуска Web-приложений с помощью Web-сервера Tomcat;
- **WebLogic Instance** — конфигурации для запуска и отладки под сервером приложений (Application Server) WebLogic.

Подробнее о настройках конфигурации запуска/отладки написано в *разделе 9.2*.

3.2.10.2. *Run (Запуск)*

"Горячие" клавиши: <Shift>+<F10>.

Команда запускает выбранную конфигурацию запуска/отладки. Подробнее о запуске приложений можно прочитать в *разделе 9.3*.

3.2.10.3. *Debug (Отладка)*

"Горячие" клавиши: <Shift>+<F9>.

Команда отладки выбранной конфигурации. Подробнее об отладке можно прочитать в *разделе 9.4*.

3.2.10.4. *Stop (Остановить)*

"Горячие" клавиши: <Ctrl>+<F2>.

Команда остановки запущенного или отлаживаемого приложения.

3.2.10.5. *Reload Changed Classes (Перезагрузить измененные классы)*

Эта команда производит перезагрузку классов, которые были изменены после запуска приложения. Эта функция также называется *HotSwap* ("горячая замена"). Данная команда работает только в режиме отладки.

Самый частый вариант использования данной команды — изменения и последующая компиляция классов во время отладки, например, когда найдена какая-либо ошибка. Выполнив эту функцию, мы "на лету" перезагружаем измененные классы и можем продолжать отладку дальше без перезапуска.

IntelliJ IDEA может автоматически перезагружать измененные классы. Чтобы это происходило, необходимо установить специальную опцию в настройках отладчика (*см. раздел 4.1.9*).

3.2.10.6. *Step Over (Шаг через)*

Команде соответствует клавиша <F8>.

Данная команда работает только при отладке. Выполнение в момент ее использования должно быть приостановлено. Эта команда позволяет выполнять программу в пошаговом режиме, не "заходя" внутрь вызовов методов.

3.2.10.7. *Step Into (Шаг внутрь)*

Команде соответствует клавиша <F7>.

Данная команда работает только при отладке. Выполнение в момент ее использования должно быть приостановлено. Эта команда позволяет выполнять программу в пошаговом режиме, "заходя" внутрь вызовов функций.

3.2.10.8. *Step Out (Шаг наружу)*

"Горячие" клавиши: <Shift>+<F8>.

Команда для отладки выполнения программы в пошаговом режиме, при этом переход происходит на один уровень выполнения вверх по стеку вызовов методов.

3.2.10.9. *Force Step Into (Принудительный шаг внутрь)*

"Горячие" клавиши: <Shift>+<F7>.

Данная команда работает аналогично команде **Step Into** (Шаг внутрь) (*см. раздел 3.2.10.7*), причем "шаг внутрь" будет произведен даже для тех методов, для которых в настройках отладки (*см. раздел 4.1.9*) установлена опция "не входить".

3.2.10.10. *Run to Cursor (Выполнять до курсора)*

"Горячие" клавиши: <Alt>+<F9>.

Команда продолжает работу программы после приостановки до тех пор, пока выполнение не дойдет до текущего положения курсора в коде.

3.2.10.11. *Pop Frame ("Вытолкнуть" окно стека)*

Команда прекращает выполнение текущего метода и переносит выполнение к точке его вызова, при этом содержимое всех локальных переменных теряется. Для программы это будет выглядеть так, как будто метод еще не выполнялся.

Внимание

Если метод уже успел изменить значения каких-либо переменных, не являвшихся локальными для этого метода, эти изменения сохранятся.

3.2.10.12. *Pause Program (Приостановить программу)*

Данная команда приостанавливает отладку программы. Команда активна в режиме отладки, когда отлаживаемая программа находится в выполняющемся состоянии.

3.2.10.13. *Resume Program (Возобновить программу)*

Команде соответствует клавиша <F9>.

Команда возобновления работы программы, находящейся в приостановленном состоянии в режиме отладки.

3.2.10.14. *Evaluate Expression (Вычислить выражение)*

"Горячие" клавиши: <Alt>+<F8>.

Данная команда открывает диалог, в котором можно, не прерывая отладку программы, вычислить некоторое выражение. Подробнее об этой возможности написано в *разделе 9.4.2*.

3.2.10.15. *Quick Evaluate Expression (Быстрое вычисление выражения)*

"Горячие" клавиши: <Alt>+<Shift>+<F8>.

Эта команда также вычисляет значение выражения, однако использует для вывода результатов вычисления всплывающее окно (Tool Tip) с результатом. Чтобы выполнить эту команду, нужно установить курсор или выделить выражение, которое необходимо вычислить, и вызвать эту команду.

Эта команда автоматически выполняется для всех переменных, полей класса и параметров вызова метода, если к ним подвести указатель мыши. Программа должна находиться под отладкой и в приостановленном режиме.

3.2.10.16. *Show Execution Point (Показать точку выполнения)*

"Горячие" клавиши: <Shift>+<F10>.

Команда подсвечивает место в коде, где находится выполнение программы в текущий момент.

3.2.10.17. *Toggle Line Breakpoint (Включить/Выключить точку останова в строке)*

"Горячие" клавиши: <Ctrl>+<F8>.

Данная команда управляет установкой и снятием точек останова в текущей строке кода. Первое выполнение команды устанавливает точку останова, второе ее снимает.

3.2.10.18. *Toggle Method Breakpoint (Включить/Выключить точку останова для метода)*

Данная команда управляет установкой и снятием точек останова для текущего метода. Первое выполнение команды устанавливает точку останова, второе ее снимает.

3.2.10.19. *Toggle Breakpoint Enabled (Включить/Выключить точку останова)*

Эта команда позволяет, не снимая точки останова, приостановить ее действие.

Внешний вид точек останова представлен на рис. 3.66.

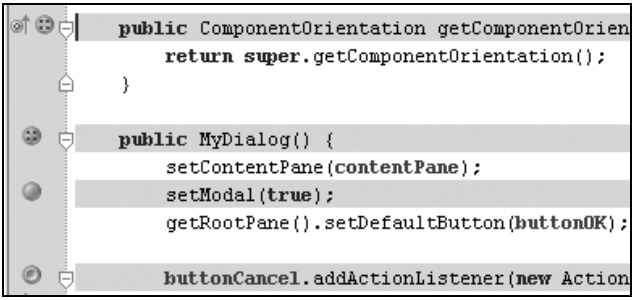


Рис. 3.66. Визуальное представление точек останова

Кружки (красного цвета) на полях редактора означают место установки точки останова. На рис. 3.66 последовательно представлены:

- выключенная точка останова для метода (на самом деле у нее посередине зеленый кружок ☺);
- точка останова для метода;
- точка останова для строки;
- выключенная точка останова для строки.

3.2.10.20. *View Breakpoints (Просмотр точек останова)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F8>.

Данная команда выводит диалог со списком всех точек останова, установленных в данном проекте (рис. 3.67).

3.2.10.21. *Export Threads (Экспорт потоков)*

Данная команда выводит в специальном диалоге список всех потоков, выполняющихся в данный момент в программе (рис. 3.68).

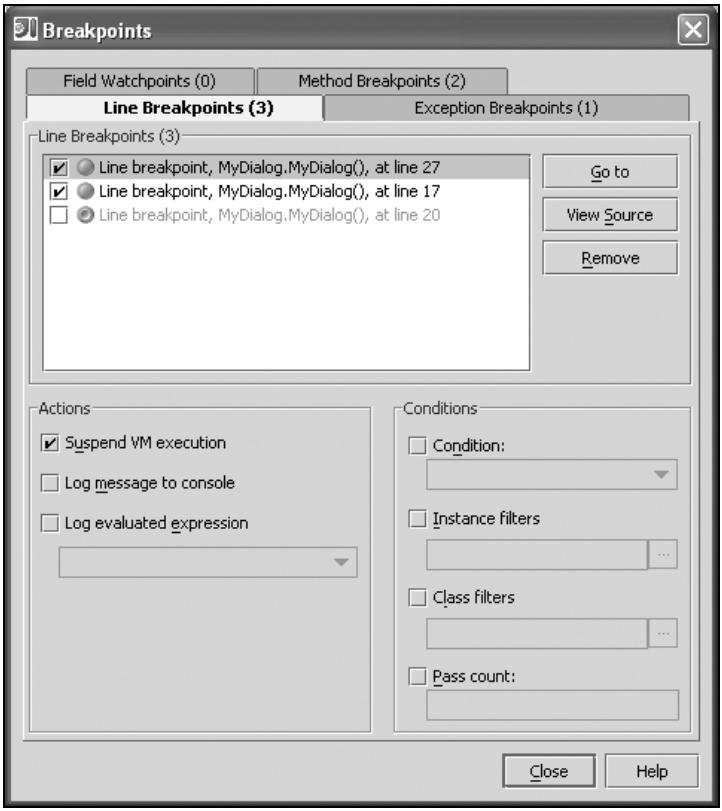


Рис. 3.67. Список точек останова

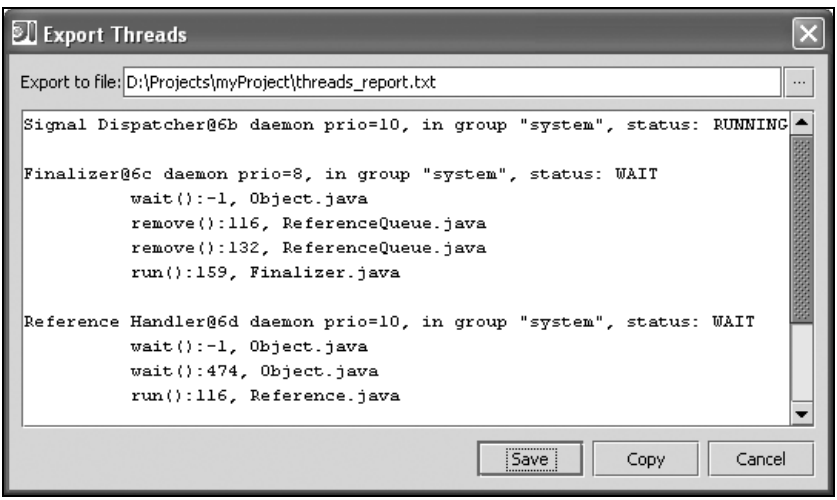


Рис. 3.68. Список потоков

Этот список можно скопировать в буфер обмена с помощью кнопки **Copy** (Копировать), а также сохранить в файл с помощью кнопки **Save** (Сохранить).

Действие этой команды аналогично нажатию сочетания клавиш <Ctrl>+<Pause> в консольном окне приложения, если оно (окно), конечно, есть.

3.2.11. Меню *Tools (Инструменты)*

В этом разделе меню собраны все команды, выполняющие вспомогательные функции, а также команды, которые достаточно сложно привязать к каким-либо определенным пунктам меню.

3.2.11.1. *Reformat Code (Переформатировать код)*

"Горячие" клавиши: <Ctrl>+<Alt>+<L>.

Данная команда переформатирует код в соответствии с текущей схемой стиля кода (рис. 3.69).

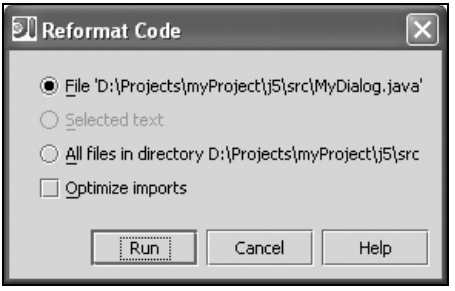


Рис. 3.69. Опции переформатирования кода

В зависимости от контекста, в котором была вызвана команда, реформат кода может быть применен:

- к выбранному файлу;
- к выделенному тексту;
- ко всем файлам в директории;
- к выбранному модулю;
- ко всему проекту.

Попутно с переформатированием можно также выполнить оптимизацию импортов, установив соответствующую опцию (**Optimize imports**).

Подробнее о схемах стиля кода можно прочитать в *разделе 4.1.7.*

3.2.11.2. *Optimize Imports (Оптимизировать импорты)*

"Горячие" клавиши: <Ctrl>+<Alt>+<O>.

Данная команда оптимизирует импорты в классах. Импорты будут оптимизированы в соответствии с настройками стиля кода (*см. раздел 4.1.7*).

3.2.11.3. *Save as Live Template (Сохранить как "живой шаблон")*

Данная команда сохраняет выделенный текст как "живой шаблон". Подробнее о настройках "живых шаблонов" написано в *разделе 4.1.13*.

3.2.11.4. *Save File as Template (Сохранить файл как шаблон)*

Команда сохраняет файл в качестве файлового шаблона. Подробнее о файловых шаблонах написано в *разделе 4.1.14*.

3.2.11.5. *Generate JavaDoc (Сгенерировать JavaDoc)*

Данная команда генерирует набор файлов JavaDoc в соответствии с настройками (рис. 3.70).

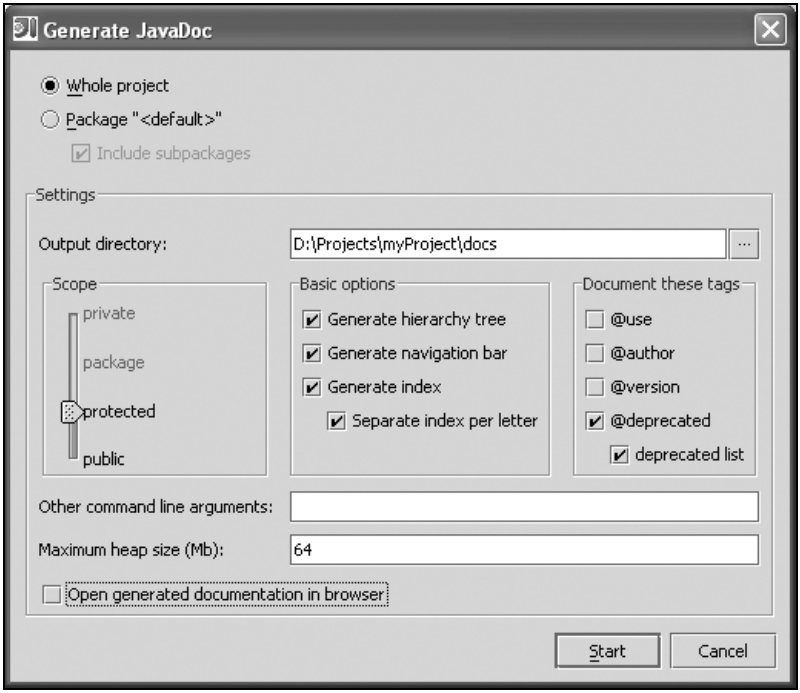


Рис. 3.70. Диалог генерации JavaDoc

Данный диалог содержит следующие настройки, которые являются визуальным отражением параметров утилиты для генерации JavaDoc:

- Область охвата файлов для генерации JavaDoc:
  - **Whole project** (Весь проект) — комментарии JavaDoc будут сгенерированы для всех классов в проекте;
  - **Package** (Пакет) — комментарии для выбранного пакета (с возможностью включения вложенных пакетов);
- **Output directory** (Выходная директория) — директория, в которой будут созданы файлы JavaDoc;
- **Scope** (Область охвата) — область охвата методов, классов и их полей по уровню доступа:
  - **private** — все методы, классы и их поля, начиная с уровня доступа `private`: `private`, `package`, `protected`;
  - **package** — все методы, классы и их поля, начиная с уровня доступа `package`: `package`, `protected`, `public`;
  - **protected** — все методы, классы и их поля, начиная с уровня доступа `protected`: `protected`, `public`;
  - **public** — только `public`-методы, классы и их поля;
- **Basic options** (Общие настройки) — настройки, относящиеся к оформлению файлов JavaDoc:
  - **Generate hierarchy tree** — генерировать дерево иерархии;
  - **Generate navigation bar** — генерировать навигационную панель;
  - **Generate index** — генерировать индекс;
  - **Separate index per letter** — отдельный индекс на каждую букву;
- **Document these tags** (Набор тегов для документирования):
  - **@use**;
  - **@author**;
  - **@version**;
  - **@deprecated**. Данная опция имеет дополнительную настройку **deprecated list** — создать список устаревших классов и методов;
- **Other command line arguments** (Прочие аргументы командной строки);
- **Maximum heap size** (Максимальный объем памяти) — размер памяти указывается в мегабайтах.

Если установить опцию **Open generated documentation in browser** (Открыть сгенерированный документ в браузере), команда автоматически откроет браузер для просмотра сгенерированной документации. О настройках браузера можно прочитать в *разделе 4.1.1*.

3.2.11.6. *Set Up JSR014 Implementation (Установить имплементацию JSR-014)*

Команда позволяет настроить JSR-014 (*см. раздел 11.1*), чтобы можно было использовать новые возможности параметризации классов (generics) из языка Java версии 5. При активации этой команды запускается мастер настройки (рис. 3.71).

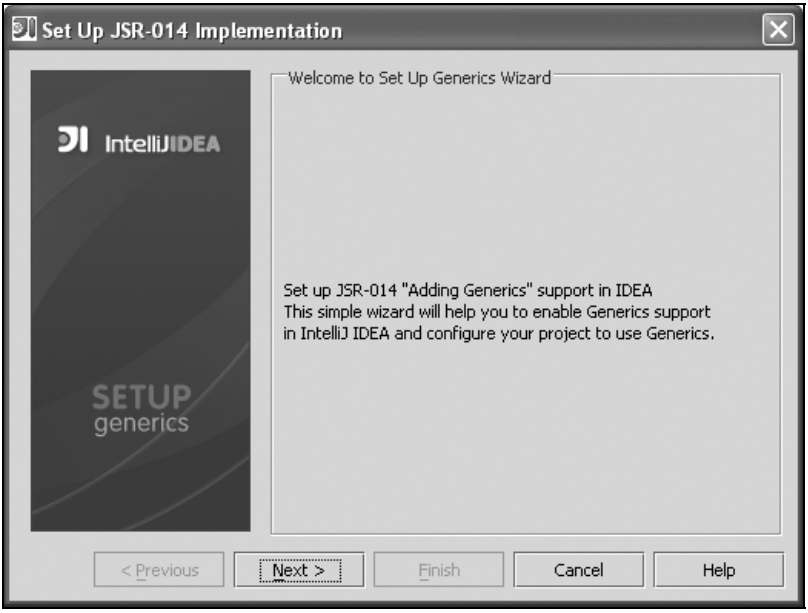


Рис. 3.71. Мастер установки JSR-014

На первом шаге мастер настройки предложит выбрать директорию, в которой установлены файлы имплементации JSR-014. В случае, если их нет, можно пойти по ссылке и скачать с сайта компании-разработчика Sun (рис. 3.72).

Затем нужно указать, для каких модулей вы хотите включить поддержку параметризации классов (рис. 3.73).

На рис. 3.73 для модуля j5 нет возможности снять выделение. Это связано с тем, что в моем проекте он сконфигурирован использовать JDK 1.5, в котором параметризованные классы поддерживаются изначально.

После нажатия кнопки **Finish** (Завершить) для выбранных модулей будет автоматически настроена поддержка параметризованных классов с помощью добавления к библиотекам модуля библиотеки collect.jar из EAP (Early Access Program — программа предварительного доступа к программному обеспечению) поставки JSR-014, которая была скачена и установлена на первом шаге.

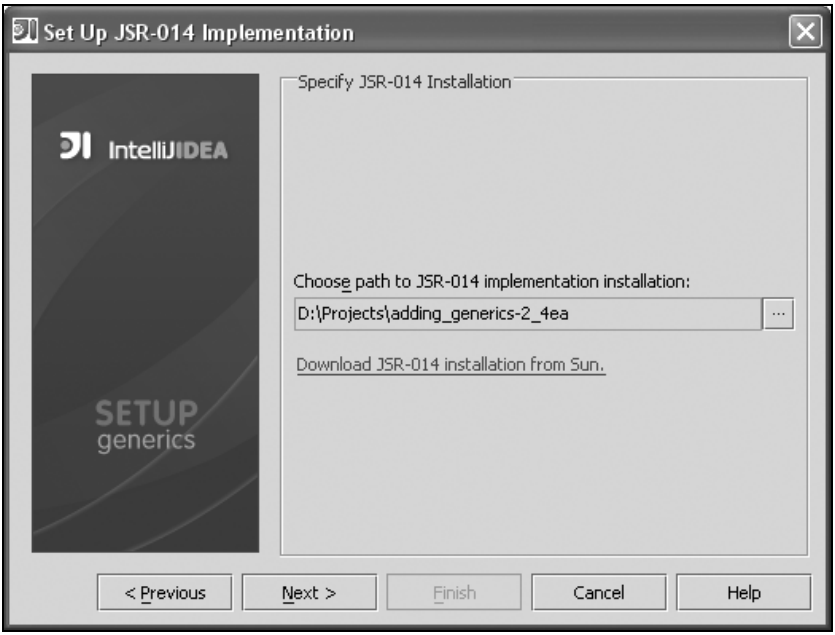


Рис. 3.72. Мастер установки JSR-014: выбор директории с имплементацией JSR-014

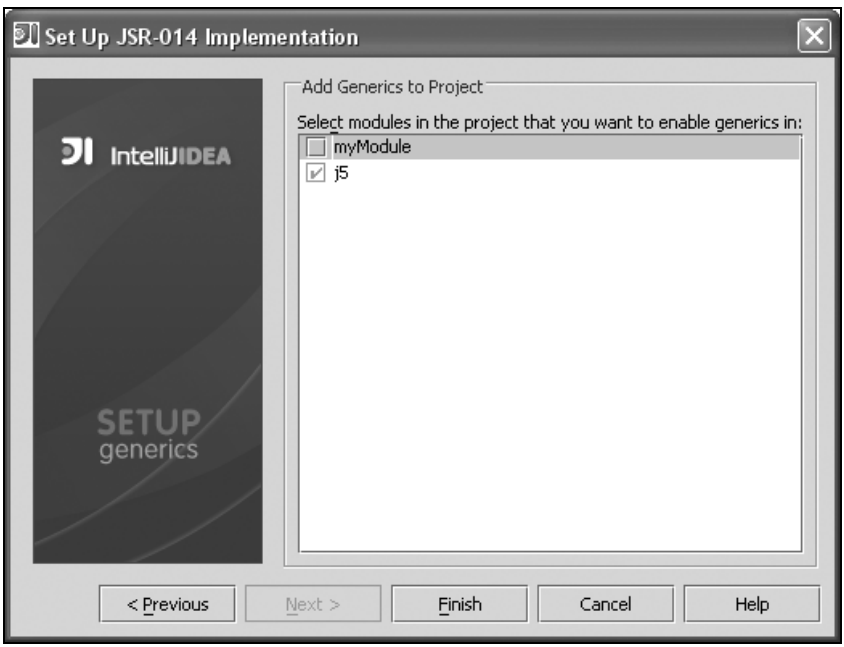


Рис. 3.73. Выбор модулей, которые используют параметризованные классы

3.2.11.7. *Macros (Макрокоманды)*

Данный пункт меню объединяет группу команд для записи, выполнения и редактирования макрокоманд. Макрокоманда — это некоторый набор из команд IntelliJ IDEA, выполненных последовательно.

Запись

Чтобы записать макрокоманду, нужно выполнить команду **Start Macro Recording** (Начать запись макрокоманды). После того, как требуемая для записи последовательность команд выполнена, нужно выполнить команду **Stop Macro Recording** (Прекратить запись макрокоманды), при этом появится диалог, в котором нужно ввести имя создаваемой макрокоманды (рис. 3.74).

Рис. 3.74. Ввод имени макрокоманды

Запуск

Созданная макрокоманда появится в меню **Macros** (Макрокоманды) (рис. 3.75). Оттуда же ее можно и запустить.

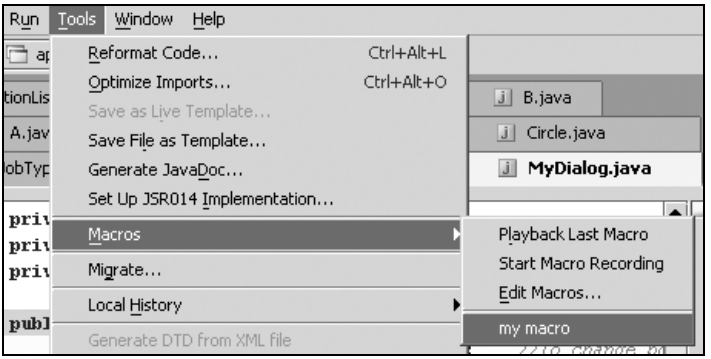


Рис. 3.75. Созданная макрокоманда в меню **Macros**

Команда **Playback Last Macro** (Проиграть последнюю макрокоманду) запустит макрокоманду, которая была запущена последней. Только что записанная макрокоманда тоже считается.

Редактирование

Выполнив команду **Edit Macros** (Редактировать макрокоманды), можно открыть диалог, в котором содержится список всех записанных макрокоманд (рис. 3.76).

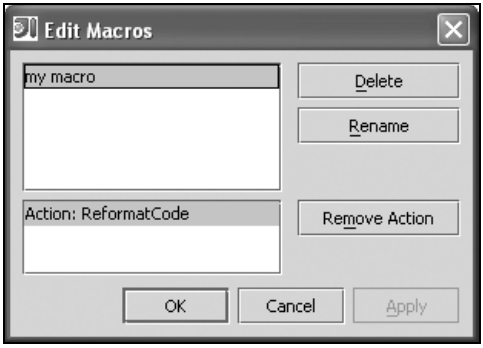


Рис. 3.76. Редактирование макрокоманд

В данном диалоге можно выполнять следующие операции с макрокомандами:

- удалять макрокоманды (**Delete**);
- переименовывать макрокоманды (**Rename**);
- удалять команды (действия) из макрокоманд (**Remove Action**).

3.2.11.8. *Migrate (Миграция)*

Данная команда позволяет осуществлять миграцию кода с одной библиотеки на другую в случае, если между названиями классов или пакетов в этих библиотеках имеется взаимно однозначное соответствие.

В текущей версии IntelliJ IDEA 4.5.2 имеется всего одна доступная миграция со Swing-версии 1.0.3 в Swing-версию 1.1 (рис. 3.77).

Также можно создавать новые миграции (**New**) и редактировать существующие (**Edit**) (рис. 3.78).

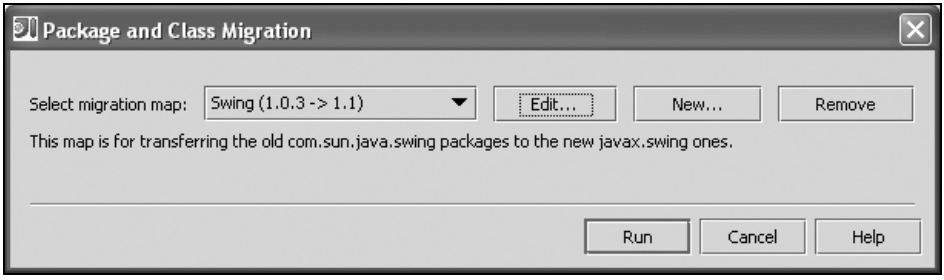


Рис. 3.77. Диалог команды миграции пакетов и классов

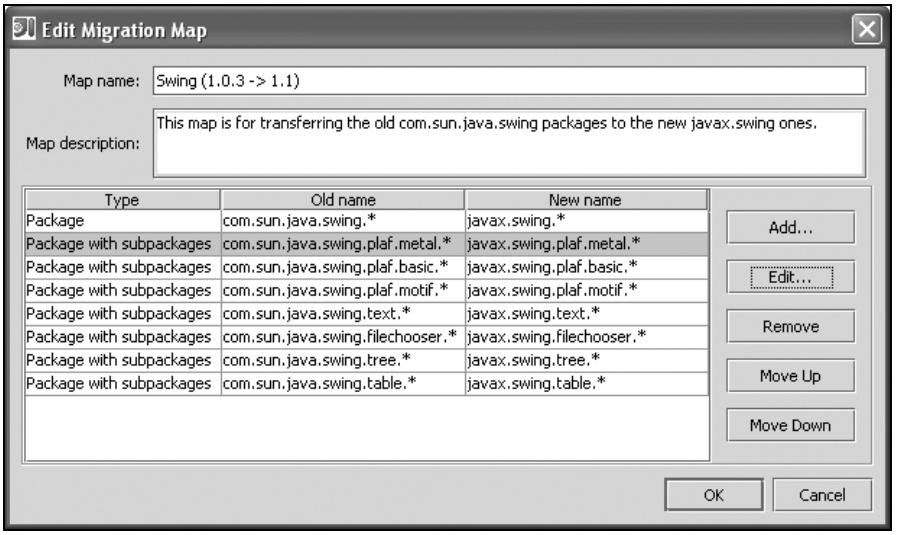


Рис. 3.78. Редактирование миграции

Для миграции можно задать следующие поля:

- ❒ название миграции (**Map name**);
- ❒ описание миграции (**Map description**);
- ❒ список старых и соответствующих им новых названий пакетов и классов.

Соответствия можно:

- ❒ добавлять (**Add**);
- ❒ редактировать (**Edit**);
- ❒ удалять (**Remove**);
- ❒ поднимать в списке вверх (**Move Up**);
- ❒ опускать в списке вниз (**Move Down**).

Добавление и редактирование соответствия производится с помощью специального диалога (рис. 3.79).

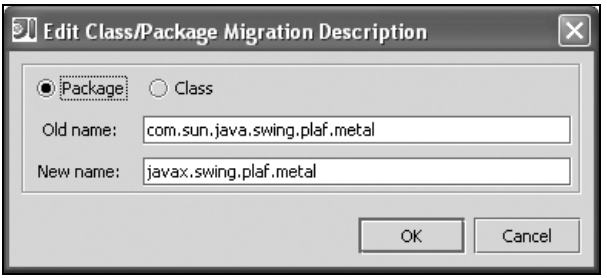


Рис. 3.79. Редактирование соответствия

Соответствия могут быть двух типов:

- для пакетов (**Package**);
- для классов (**Class**).

Кроме типа для соответствия нужно задать старое имя (**Old name**) и новое имя (**New name**) пакета или класса. В именах пакетов можно использовать маску * для обозначения всех классов из данного пакета.

Команда миграции действует для всех классов в проекте.

3.2.11.9. *Local History (Локальная история)*

Данная команда позволяет управлять локальной системой контроля версий, встроенной в IntelliJ IDEA. Подробно эта возможность описана в *разделе 12.2*.

3.2.11.10. *Generate DTD from XML file (Сгенерировать DTD для XML-файла)*

Данная команда создает для XML-файла встроенное описание DTD (Document Type Definition — определение типа документа). Рассмотрим работу этой команды на примере. Допустим, у нас есть XML-файл, представленный в листинге 3.6.

Листинг 3.6

```
<materials>
    <material id="11" title="Wood"/>
    <material id="12" title="Stone"/>
    <material id="13" title="Glue"/>

    <compatibility>
        <material idref="11"/>
        <material idref="13"/>
    </compatibility>

    <compatibility>
        <material idref="12"/>
        <material idref="13"/>
    </compatibility>
</materials>
```

Если применить команду **Generate DTD from XML file** (Сгенерировать DTD для XML-файла) к файлу из листинга 3.6, в результате мы получим то, что представлено в листинге 3.7.

Листинг 3.7

```
<!DOCTYPE materials [
<!ELEMENT material (#PCDATA)>
<!ATTLIST material
    id ID #IMPLIED
    idref CDATA #IMPLIED
    title CDATA #IMPLIED>
<!ELEMENT materials (material|compatibility)*>
<!ELEMENT compatibility (material)*>
]><materials>
    <material id="11" title="Wood"/>
    <material id="12" title="Stone"/>
    <material id="13" title="Glue"/>

    <compatibility>
        <material idref="11"/>
        <material idref="13"/>
    </compatibility>

    <compatibility>
        <material idref="12"/>
        <material idref="13"/>
    </compatibility>
</materials>
```

Данная команда не отличается большой сообразительностью и не вдается в тонкости структуры XML-документа.

Можно заметить, что результат получен с учетом некоторых предположений. Например, значения атрибута `title` элемента `material` имеют строго ограниченные названия, однако команда посчитала их просто некоторым текстом. Это предположение не является ошибочным, но, возможно, мы захотим изменить его на другое, например, такое: `title (Wood | Stone| Glue) #IMPLIED`. Или определение элемента `material`. Его корректнее было бы описать как `<!ELEMENT material EMPTY>`. Но это уже дается на "откуп" программисту.

Данную команду удобно использовать, работая с XML-файлами неизвестной структуры. Для них можно очень быстро получить примерное DTD-описание, из которого структура XML-файла становится видна как на ладони.

3.2.11.11. *Inspection Gadgets error highlighting (Подсветка ошибок для Inspection Gadgets)*

"Горячие" клавиши: <Alt>+<I>.

Данная команда позволяет включать и выключать подсветку ошибок, найденных инспекционным плагином Inspection Gadgets, поставляемым вместе с IntelliJ IDEA. Подробнее о настройки подсветки ошибок можно прочитать в *разделе 4.1.5*. Подробнее о плагинах можно прочитать в *главе 15*.

3.2.12. Меню *Window (Окно)*

Команды из данного раздела меню служат для управления окнами IntelliJ IDEA. Как и многие разделы меню, раздел **Window** (Окно) разделен на логические группы.

3.2.12.1. Команды инструментальных окон

Первой идет группа команд, отвечающих за визуализацию инструментальных окон (ToolWindow). Команды из этой группы активны тогда, когда инструментальные окна, с которыми они связаны, уже присутствуют в основном окне IntelliJ IDEA. Подробнее об инструментальных окнах рассказано в *разделе 3.4*, здесь же мы приведем лишь их краткий перечень:

- **Messages** (Сообщения) <Alt>+<0> — окно с различными сообщениями;
- **Project** (Проект) <Alt>+<1> — окно со структурой проекта;
- **Commander** (Коммандер) <Alt>+<2> — окно Коммандера;
- **Find** (Найти) <Alt>+<3> — окно с результатами поиска;
- **Run** (Запуск) <Alt>+<4> — окно с консолью запущенного приложения;
- **Debug** (Отладка) <Alt>+<5> — окно с информацией об отладке приложения;
- **TODO** (Заметки) <Alt>+<6> — окно со списком всех комментариев, содержащих слово "TODO" (от англ. "to do" — сделать);
- **Structure** (Структура) <Alt>+<7> — окно с внутренней структурой текущего файла;
- **Hierarchy** (Иерархия) <Alt>+<8> — окно с иерархией;
- **Ant Build** (Сборка утилитой Ant) — окно с командами из сборочного файла утилиты Ant;
- **CVS** — окно результатов работы системы контроля версий CVS. Данная команда добавляется плагином поддержки CVS;

- **Dependency Viewer** (Просмотр зависимостей) — окно результата поиска зависимостей;
- **Inspection** (Инспекции) — окно результата работы инспекции.

Следующая группа содержит две команды.

3.2.12.2. *Store Current Layout as Default/Restore Default Layout (Сохранить текущее расположение в качестве расположения по умолчанию/ Восстановить расположение по умолчанию)*

Данная пара команд позволяет сохранять текущее расположение всех окон, открытых в IntelliJ IDEA, делая его расположением по умолчанию, и восстанавливать это расположение.

3.2.12.3. *Hide Active Window (Спрятать активное окно)*

"Горячие" клавиши: <Shift>+<Escape>.

Команда прячет активное инструментальное окно.

3.2.12.4. *Hide All Windows (Спрятать все окна)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F12>.

Команда прячет все инструментальные окна, открытые в данный момент.

3.2.12.5. *Jump to Last Window (Перейти к последнему окну)*

"Горячие" клавиши: <F12>.

Команда открывает последнее инструментальное окно, которое было активным.

3.2.12.6. *Pinned Mode (Статус "кнопки")*

Команда переключает статусы инструментального окна между прячущимся статусом и статусом постоянного отображения.

3.2.12.7. *Docked Mode (Режим привязки)*

Команда переключает состояния инструментального окна между состоянием перекрытия и состоянием смещения.

3.2.12.8. *Floating Mode (Плавающий режим)*

Команда переключает режим отображения инструментального окна между плавающим и фиксированным режимом.

3.2.12.9. *Select Next/Previous Tab (Выбрать следующую/предыдущую закладку)*

"Горячие" клавиши: <Alt>+<Right>/<Alt>+<Left>.

Команда открывает окно редактора, находящееся в следующей/предыдущей закладке после/перед текущей активной закладки.

Переход между закладками зациклен, поэтому с помощью данных команд можно перейти к любой интересующей вас закладке с редактором. Подробнее о работе с закладками редактора написано в *разделе 3.6*.

3.2.12.10. *Close Active Tab (Закрыть активную закладку)*

"Горячие" клавиши: <Ctrl>+<Shift>+<F4>.

Команда закрывает активную закладку с редактором.

3.2.12.11. *Pin Active Tab (Закрепить активную закладку)*

Команда позволяет закрепить активную закладку так, чтобы она всегда отображалась среди закладок с редакторами до тех пор, пока не будет принудительно закрыта.

3.2.13. Меню *Help (Помощь)*

Данный раздел меню содержит вполне стандартный набор команд, который можно встретить в любом приложении.

3.2.13.1. *Help Topics (Разделы помощи)*

Данная команда открывает окно встроенной справочной системы IntelliJ IDEA (рис. 3.80).

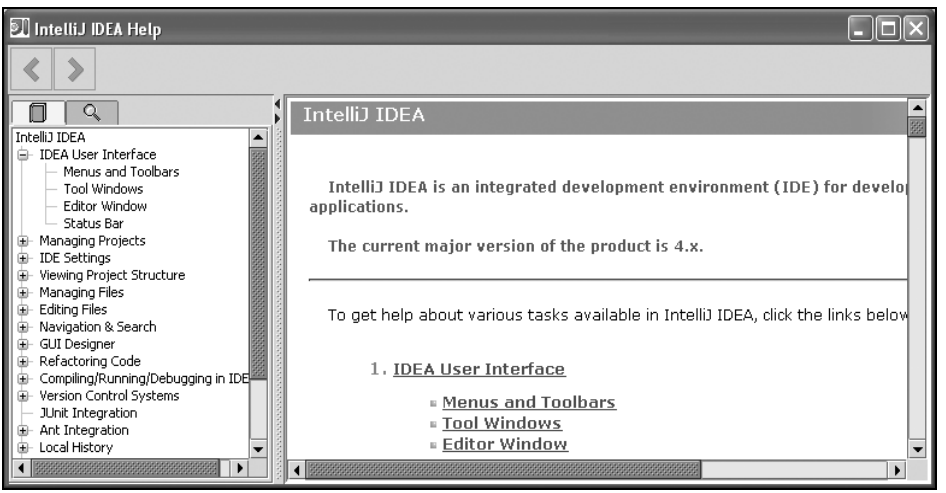


Рис. 3.80. Внешний вид окна справочной системы

Текст справочной информации написан на английском языке. Кроме навигации по содержанию справочной информации, имеется также и поиск (рис. 3.81).

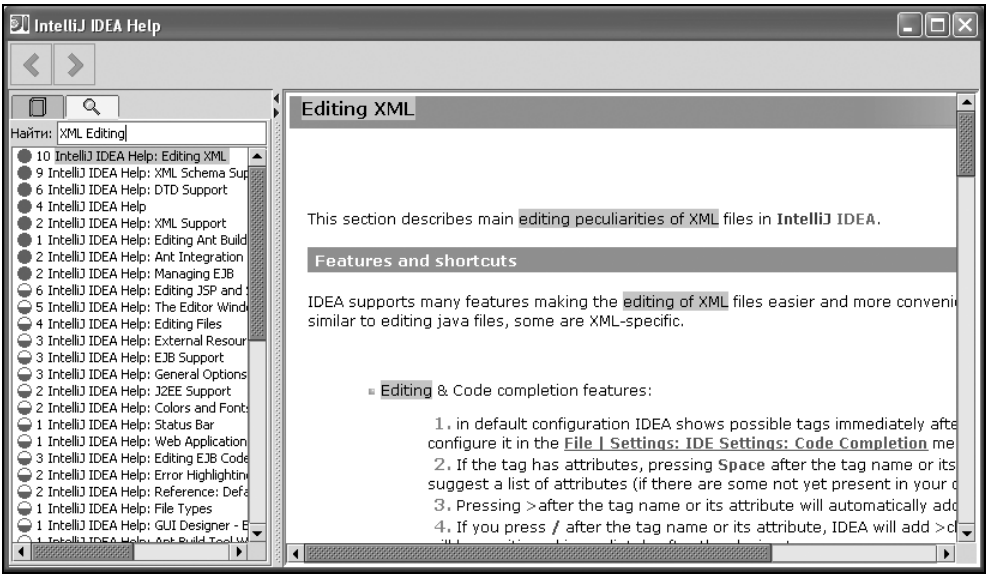


Рис. 3.81. Результаты поиска информации в справочной системе

3.2.13.2. *Tip of the Day (Подсказка дня)*

Данная команда отображает диалог с короткой справочной информацией, посвященной различным аспектам работы с IntelliJ IDEA (рис. 3.82).

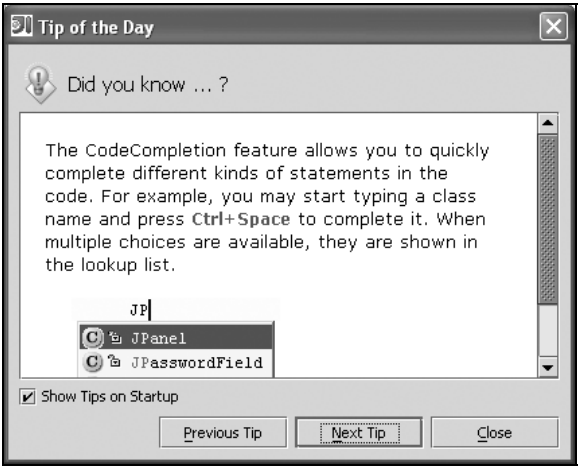


Рис. 3.82. Результат работы команды **Tip of the Day**

Есть несколько таких подсказок, которые можно просматривать последовательно, нажимая кнопки **Previous Tip** (Предыдущая подсказка) и **Next Tip** (Следующая подсказка).

Установив опцию **Show Tips on Startup** (Показывать подсказки при старте), можно будет начинать работу с IntelliJ IDEA с такого окна (весьма полезно для начинающих пользователей IntelliJ IDEA).

3.2.13.3. *Productivity Guide (Руководство по продуктивности)*

Эта команда появилась в IntelliJ IDEA совсем недавно, в версии 4.5. Ее суть заключается в сборе статистической информации о тех командах, которые выполняет пользователь по мере работы с IntelliJ IDEA (рис. 3.83).

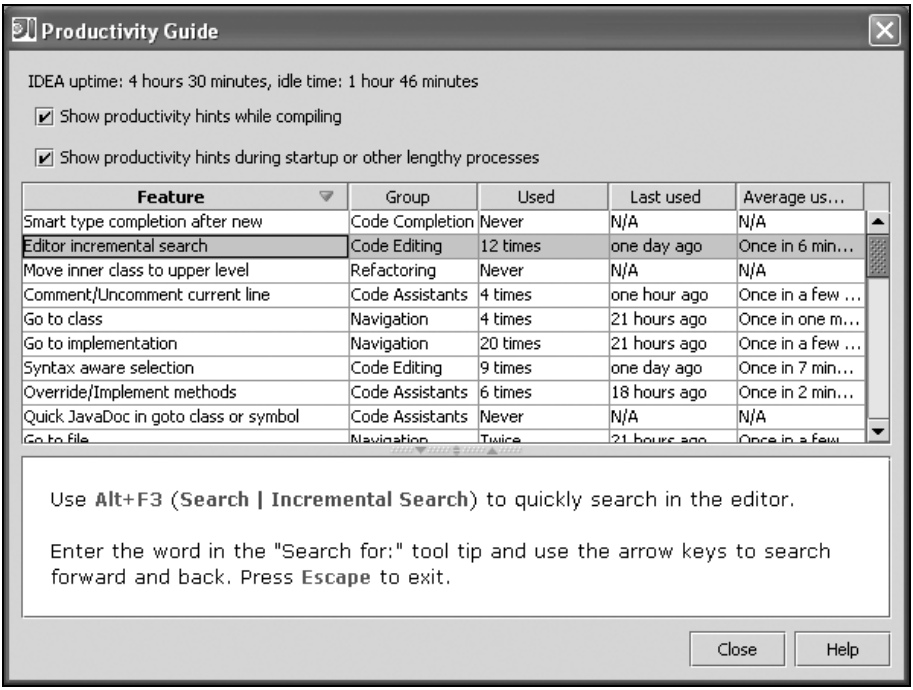


Рис. 3.83. Статистика продуктивности работы программиста

На основе собранной статистики IntelliJ IDEA делает заключение о том, насколько продуктивна работа программиста, и нельзя ли ее каким-либо образом улучшить. Выводы о продуктивности делаются довольно просто. Если пользователь не использует или использует редко какую-либо команду, которую IntelliJ IDEA считает важной, IntelliJ IDEA будет советовать пользоваться ею чаще, выводя соответствующие подсказки в момент длительных операций (рис. 3.84).

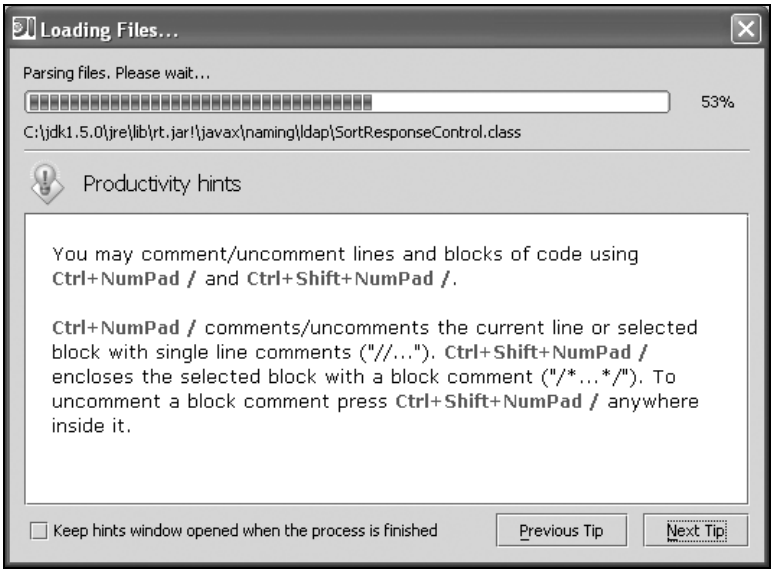


Рис. 3.84. Подсказка о повышении производительности труда

С помощью кнопок **Previous Tip** (Предыдущая подсказка) и **Next Tip** (Следующая подсказка) можно просмотреть все подсказки о повышении продуктивности.

Выбрав опцию **Keep hints window opened when the process is finished** (Оставить окно с подсказками после завершения процесса), можно оставить окно с подсказками даже после того, как длительная операция закончится.

В диалоге со статистикой о продуктивности есть две опции, указывающие, когда должны показываться подсказки о продуктивности:

- **Show productivity hints while compiling** — показывать подсказки о продуктивности во время компиляции;
- **Show productivity hints during startup or other lengthy processes** — показывать подсказки о продуктивности во время запуска IntelliJ IDEA, а также во время любых длительных операций.

В верхней части диалога со статистикой содержится информация о суммарном времени активной работы IntelliJ IDEA, а также времени простоя.

3.2.13.4. *Online Documentation (Онлайн-документация)*

Данная команда открывает браузер с сайтом фирмы-разработчика JetBrains **http://www.jetbrains.com**, на котором расположена различная документация по IntelliJ IDEA.

О настройках браузера подробнее можно прочитать в *разделе 4.1.1.*

3.2.13.5. *Register (Регистрация)*

IntelliJ IDEA — коммерческий продукт, для работы которого требуется специальный лицензионный ключ, который можно ввести в диалоге, открываемом этой командой (рис. 3.85).

Есть три способа получить лицензионный ключ:

1. Зарегистрироваться на сайте JetBrains **http://www.jetbrains.com** и получить временную лицензию на 30 дней для знакомства с продуктом. Лицензионный ключ при этом будет прислан вам по электронной почте.
2. Купить лицензию.
3. Использовать лицензию по программе предварительного доступа.

Подробнее о лицензионных соглашениях использования IntelliJ IDEA можно прочитать в *главе 2*.

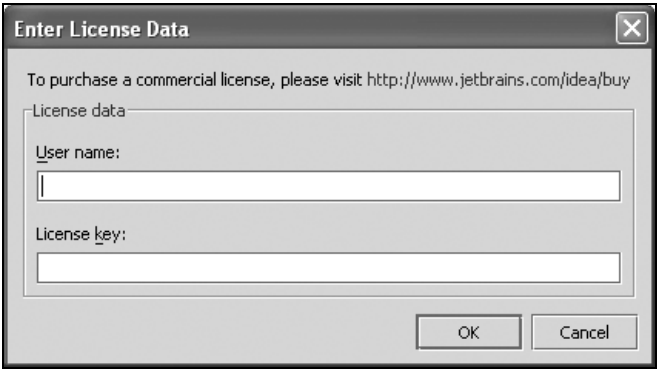


Рис. 3.85. Регистрация IntelliJ IDEA

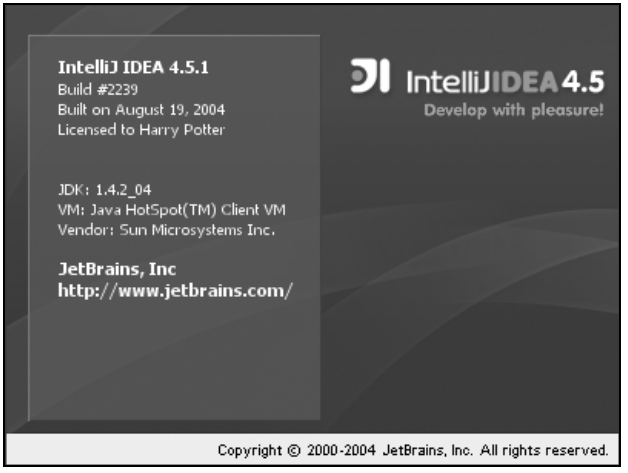


Рис. 3.86. Сведения о программе

3.2.13.6. *About (О программе)*

Данная команда выводит диалог с информацией о версии IntelliJ IDEA, порядковом номере установленной сборки (build number), дате создания дистрибутива, а также о версии JVM (Java Virtual Machine — Виртуальная машина Java), под которой работает IntelliJ IDEA (рис. 3.86).

3.3. Панель инструментов (*Tool Bar*)

В интерфейсе IntelliJ IDEA есть главная панель инструментов, расположенная в верхней части основного окна (см. рис. 3.1, 3.87).

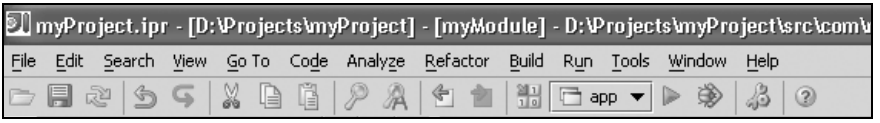


Рис. 3.87. Панель инструментов

В панели инструментов располагаются кнопки, связанные с наиболее часто используемыми командами. Всего таких команд восемнадцать:

- **Open File** (Открыть файл) — команда из меню **File** (Файл) открывает файл в редакторе IntelliJ IDEA (*см. раздел 3.2.1.4*);
- **Save All** (Сохранить все) — команда из меню **File** (Файл) сохраняет все измененные документы в IntelliJ IDEA (*см. раздел 3.2.1.13*);
- **Synchronize** (Синхронизировать) — команда из меню **File** (Файл) находит все файлы, которые были изменены извне, и перезагружает их (*см. раздел 3.2.1.16*);
- **Undo** (Отменить) — команда из меню **Edit** (Редактирование) отменяет последнее сделанное изменение (*см. раздел 3.2.2.1*);
- **Redo** (Повторить) — команда из меню **Edit** (Редактирование) повторяет последнее отмененное изменение (*см. раздел 3.2.2.2*);
- **Cut** (Вырезать) — команда из меню **Edit** (Редактирование) удаляет класс, файл или текст в редакторе в буфер обмена (*см. раздел 3.2.2.3*);
- **Copy** (Копировать) — команда из меню **Edit** (Редактирование) копирует в буфер обмена файл, класс или текст в редакторе (*см. раздел 3.2.2.4*);
- **Paste** (Вставить) — команда из меню **Edit** (Редактирование) вставляет из буфер обмена то, что туда было помещено командой **Copy** (Копировать) или **Cut** (Вырезать) (*см. раздел 3.2.2.6*);
- **Find** (Найти) — команда из меню **Search** (Поиск) ищет строку в тексте. Работает как в окне редактора, так и в любом другом окне, содержащем текст, например в консоли запущенного приложения (*см. раздел 3.2.3.1*);

- **Replace** (Заменить) — команда из меню **Search** (Поиск) заменяет строку в тексте (*см. раздел 3.2.3.2*). Работает в редакторе;
- **Back** (Назад) — команда из меню **Go To** (Перейти) перемещает курсор к предыдущему месту в файле (*см. раздел 3.2.5.12*). Работает в окнах редактора;
- **Forward** (Вперед) — команда из меню **Go To** (Перейти) перемещает курсор к следующему месту в файле (*см. раздел 3.2.5.12*). Работает в окнах редактора;
- **Make Project** (Собрать проект) — команда из меню **Build** (Сборка) выполняет сборку проекта — компилирует файлы, которые еще не были скомпилированы либо изменились с момента последней компиляции (*см. раздел 3.2.9.2*);
- **Select Run/Debug Configurations** (Выбрать конфигурацию Запуска/Отладки) (<Alt>+<Shift>+<F10>) — данная команда представлена выпадающим списком доступных конфигураций для запуска и отладки (*см. раздел 3.2.10.1*). В конце списка имеется команда, дублирующая команду **Edit Configurations** (Редактировать конфигурации) из меню **Run** (Запуск). Справа от данного выпадающего списка расположены кнопки для запуска и отладки выбранной конфигурации в списке;
- **Run** (Запуск) — команда из меню **Run** (Запуск) запускает выбранную конфигурацию (*см. раздел 3.2.10.2*);
- **Debug** (Отладка) — команда из меню **Run** (Запуск) запускает отладку с выбранной конфигурацией (*см. раздел 3.2.10.3*);
- **Settings** (Настройки) — команда из меню **File** (Файл) открывает диалог со всеми возможными настройками IntelliJ IDEA (*см. раздел 3.2.1.11*);
- **Help Topics** (Разделы помощи) — команда из меню **Help** (Помощь) открывает окно встроенной справочной системы IntelliJ IDEA (*см. раздел 3.2.13.1*).

К сожалению, IntelliJ IDEA не позволяет программисту самостоятельно выбирать команды, отображаемые на панели инструментов, возможно, эта функция появится в будущих версиях IntelliJ IDEA.

3.4. Инструментальные окна (*Tool Window*)

Все инструментальные окна в IntelliJ IDEA имеют стандартный вид (см. рис. 3.2, 3.88).

Инструментальные окна связаны с кнопками вызова, располагающимися на полях, по краю основного окна IntelliJ IDEA.

Инструментальные окна хороши тем, что могут находиться в любом удобном месте экрана, занимают мало места и могут автоматически исчезать и появляться при необходимости.

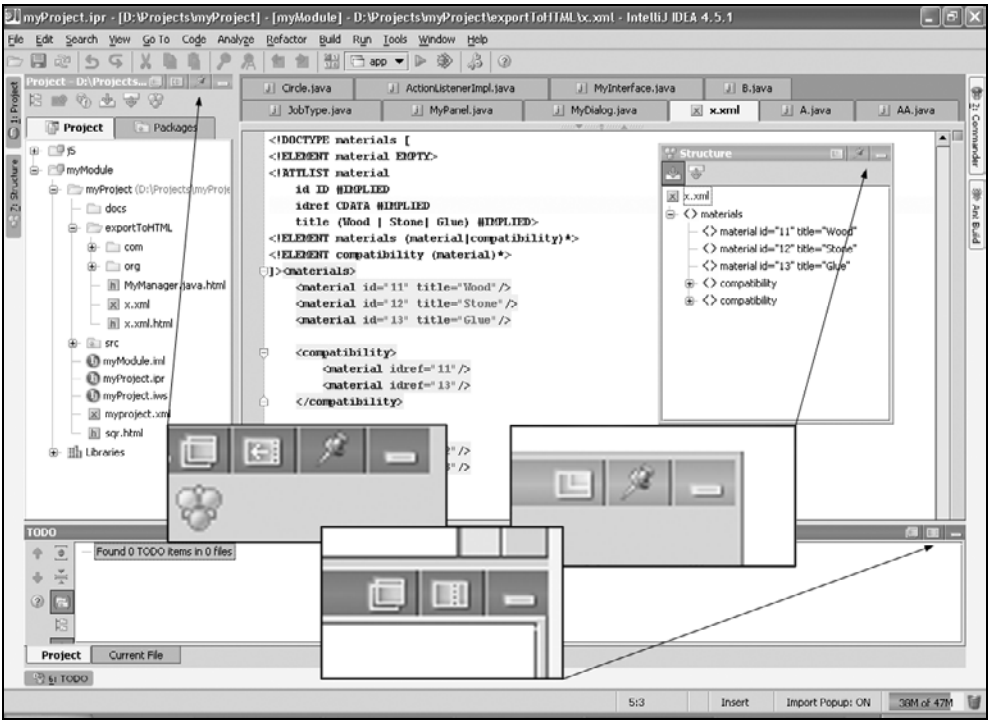


Рис. 3.88. Различные способы отображения инструментальных окон

3.4.1. Режимы отображения

Существует два основных режима отображения инструментальных окон:

- плавающий режим — окно может располагаться в произвольном месте в пределах основного окна IntelliJ IDEA;
- фиксированный режим — окно имеет фиксированное размещение рядом с одной из границ основного окна.

На рис. 3.2 все три инструментальных окна имеют фиксированное расположение у левой, нижней и правой границ основного окна. На рис. 3.88 два инструментальных окна имеют фиксированное расположение у левой и нижней границы, третье находится в плавающем режиме.

Любое инструментальное окно имеет в верхнем правом углу несколько управляющих кнопок (три или четыре, в зависимости от режима отображения).

За переключение в плавающий режим и соответственно для выхода из него в фиксированный отвечает крайняя левая кнопка. В левом увеличенном фрагменте на рис. 3.88 — это кнопка **Float** (Плавать), в правом увеличенном фрагменте — это кнопка **Fix** (Фиксировать).

3.4.2. Состояния окна

Когда инструментальное окно находится в фиксированном режиме (привязано к одной из границ основного окна), у него имеется два дополнительных состояния:

- состояние перекрытия — окно перекрывает содержимое окон редактора под собой;
- состояние смещения — окно сдвигает окно редактора.

Переключением между этими двумя состояниями заведует вторая слева кнопка (левый и нижний увеличенные фрагменты на рис. 3.88). В левом увеличенном фрагменте — это кнопка **Undock** (Перекрывать), в нижнем фрагменте — **Dock** (Смещать). Инструментальное окно, расположенное слева, находится в состоянии смещения. Окно снизу — в состоянии перекрытия.

3.4.3. Статусы окна

Также имеется два дополнительных статуса окон, находящихся в плавающем режиме или в фиксированном режиме в состоянии перекрытия:

- прячущийся статус — в нем окно присутствует на экране до тех пор, пока является активным (заголовок окна темного цвета), т. е. когда оно имеет фокус ввода. Как только окно теряет фокус, оно тут же прячется с глаз долой и перестает занимать столь ценное место на экране;
- статус постоянного отображения — в нем окно все время будет видно на экране, даже если не будет обладать фокусом ввода.

Переключением между этими двумя статусами заведует вторая справа кнопка (левый и правый увеличенные фрагменты на рис. 3.88) — кнопка в виде "кнопки", которую используют, чтобы приколоть лист бумаги к доске объявлений.

Когда инструментальное окно имеет прячущийся статус, эта кнопка называется **Pin** (Приколоть). Для окна в статусе постоянного отображения (окно у левой границы основного окна, а также плавающее окно, см. рис. 3.88) — это кнопка **Unpin** (Открепить).

Последняя крайняя правая кнопка отвечает за сворачивание инструментального окна вне зависимости от режима его отображения.

3.4.4. Размещение кнопок вызова

Кнопки вызова инструментальных окон располагаются на полях основного окна по его периметру. Для того чтобы изменить расположение этих кнопок, а также привязку к той или иной границе окна соответственно, можно использовать четыре специальные команды. Эти команды объединены разделом меню **Move To** (Переместить), который находится в контекстном меню

каждого инструментального окна. Для доступа к контекстному меню окна можно использовать нажатие правой кнопки мыши на заголовке инструментального окна. В разделе меню **Move To** (Переместить) находятся следующие команды:

- **Top** (Верх) — выполняет привязку инструментального окна к верхней границе основного окна IntelliJ IDEA;
- **Bottom** (Низ) — выполняет привязку инструментального окна к нижней границе основного окна;
- **Left** (Лево) — выполняет привязку инструментального окна к левой границе основного окна;
- **Right** (Право) — выполняет привязку инструментального окна к правой границе основного окна.

3.4.5. Дополнительные команды управления

Каждое инструментальное окно в IntelliJ IDEA может иметь собственную комбинацию клавиш для быстрого вызова. Десять базовых инструментальных окон, из описанных в *разделе 3.4.8*, имеют стандартную комбинацию клавиш вызова вида <Alt>+<цифра> (<цифра> — клавиши от 0 до 9).

3.4.6. Настройки инструментальных окон

Настройка общих параметров инструментальных окон производится с помощью диалога настройки **Appearance** (Внешний вид) (*см. раздел 4.1.2*).

3.4.7. Поиск в инструментальных окнах

Во многих инструментальных окнах работает функция быстрого поиска, чем-то похожая на функцию инкрементального поиска (*см. раздел 3.2.3.6*).

Если начать набирать на клавиатуре символы, появится небольшая панель, в которой будет отображаться ввод. Набранный текст автоматически ищется среди элементов структуры инструментального окна, и, если соответствующие элементы будут найдены, они будут подсвечены.

В отличие от инкрементального поиска, в строке поиска можно использовать символ * в качестве маски, обозначающей произвольный набор символов.

3.4.8. Базовые инструментальные окна

Все базовые инструментальные окна в IntelliJ IDEA имеют схожее строение (см. рис. 3.88).

Практически все базовые инструментальные окна имеют дополнительную панель инструментов (ToolBar) либо в левой, либо верхней части окна (ис-

ключение составляет только **Commander** (Коммандер) — *см. раздел 3.4.8.3*). Набор кнопок, которые располагаются на панели инструментов, различен для каждого окна и также может отличаться в зависимости от того, какая команда использует инструментальное окно для вывода информации.

Если инструментальное окно позволяет представлять информацию разного типа, либо полученную от разных команд, оно может снабжаться закладками (см. рис. 3.90). Информация от каждой команды будет представлена на собственной закладке.

Далее мы рассмотрим все базовые инструментальные окна в IntelliJ IDEA. Если инструментальное окно имеет отдельную функциональность, проявляющуюся при работе какой-либо команды, такая особенность будет описана в разделе, посвященном конкретной команде.

3.4.8.1. *Messages (Сообщения)*

"Горячие" клавиши: <Alt>+<0>.

Это окно используется для вывода сообщений, возникших по ходу работы различных команд IntelliJ IDEA. Например, здесь выводятся сообщения при компиляции или при запуске команд сборки утилиты Ant. Общий вид инструментального окна **Messages** (Сообщения) представлен на рис. 3.89. Сообщения могут быть некоторым образом сгруппированы.

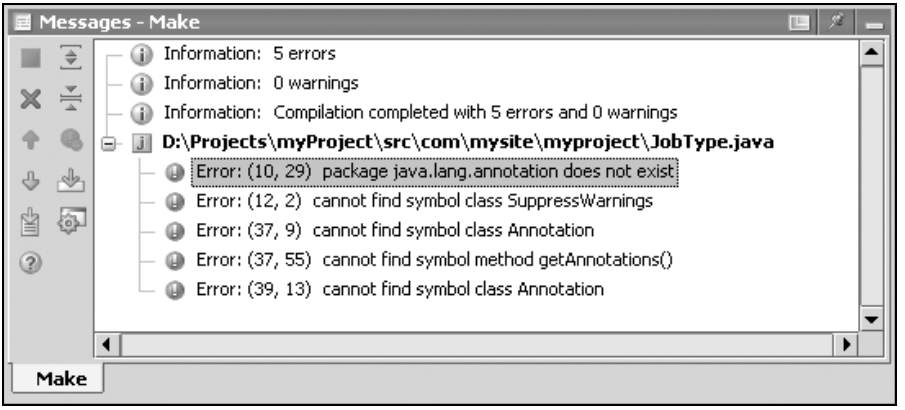


Рис. 3.89. Инструментальное окно **Messages**

В панель инструментов данного окна входят следующие команды, доступные при просмотре любых сообщений:

- **Next/Previous Message** (Следующее/Предыдущее сообщение) <Ctrl>+<Alt>+<Up>/<Ctrl>+<Alt>+<Down> — кнопки со стрелками <↑> и <↓> — переход к предыдущему/следующему сообщению в списке;

- **Help** (Помощь) <F1> — вывод контекстной справочной информации о работе данного инструментального окна;
- **Expand/Collapse All** (Развернуть/Свернуть все) <Ctrl>+<+>/<Ctrl>+<-> — развернуть/свернуть все группировки сообщений.

3.4.8.2. *Project (Проект)*

"Горячие" клавиши: <Alt>+<1>.

Инструментальное окно **Project** (Проект) используется для представления иерархической структуры проекта (рис. 3.90).

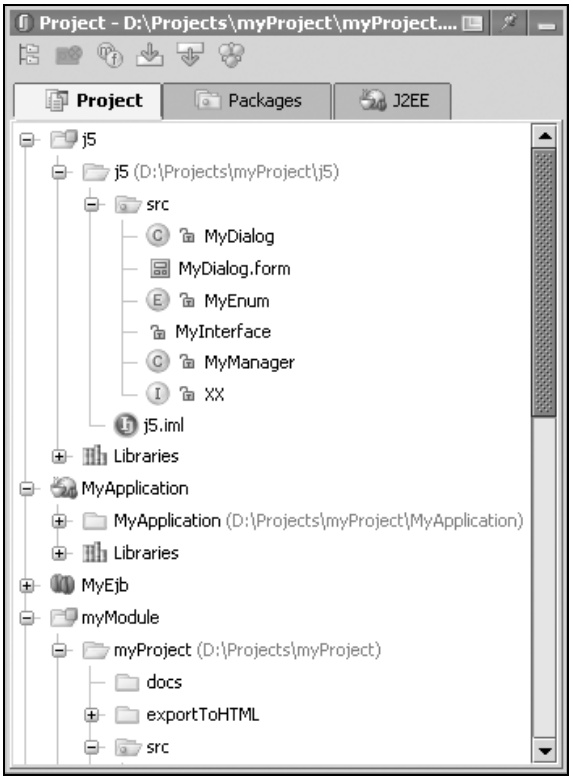


Рис. 3.90. Инструментальное окно **Project**

Окно имеет несколько закладок:

- **Project** (Проект) — данная закладка присутствует в любых проектах. В ней отображается общая структура проекта;
- **Packages** (Пакеты) — данная закладка присутствует в любых проектах. В ней отображается структура пакетов;

- **J2EE** — данная закладка опциональна и появляется, если в проекте имеются Web-модули, J2EE-Application-модули или EJB-модули (подробнее о них можно прочитать в *главах 5, 10*).

В верхней части окна расположена инструментальная панель (ToolBar) со следующими общими командами:

- **Flatten Packages** (Плоские пакеты) — команда переключает режим просмотра пакетов с исходными файлами проекта. Если опция выключена, структура пакетов отображается полностью иерархически (рис. 3.91). Если опция включена, пакеты записываются полными именами и располагаются в виде списка (рис. 3.92);

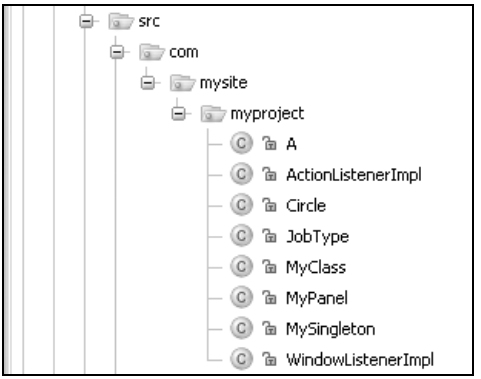


Рис. 3.91. Иерархическое представление пакетов

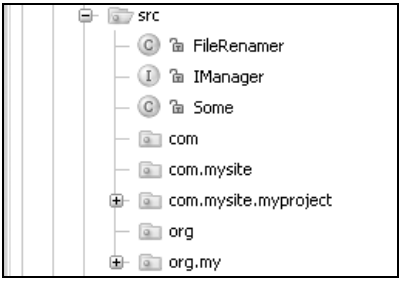


Рис. 3.92. Плоское представление пакетов

- **Hide Empty Middle Packages** (Скрывать пустые промежуточные пакеты) — команда, доступная только при плоском режиме отображения пакетов. Позволяет скрывать в списке пустые пакеты вроде com, com.mysite и org, как на рис. 3.92;
- **Show Members** (Показывать членов класса) — команда, позволяющая для каждого класса из структуры проекта показывать его структуру из методов и полей (рис. 3.93). Данная структура аналогична той, которую можно

посмотреть для класса в инструментальном окне **Structure** (Структура) (*см. раздел 3.4.8.8*);

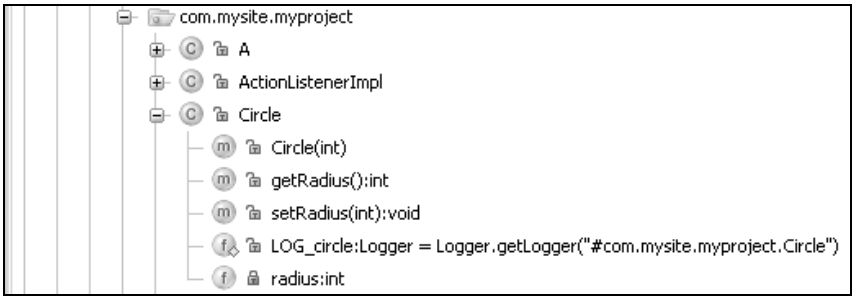


Рис. 3.93. Отображение членов класса

- **Autoscroll to Source** — команда, позволяющая автоматически выводить в окне редактора содержимое файлов, классы которых просматриваются в инструментальном окне;
- **Autoscroll from Source** — команда, обратная предыдущей. Позволяет автоматически перемещать содержимое инструментального окна в соответствии с тем, какой файл редактируется;
- **Show structure** (Показывать структуру) — команда дополняет нижнюю часть инструментального окна **Project** (Проект) окном **Structure** (Структура).

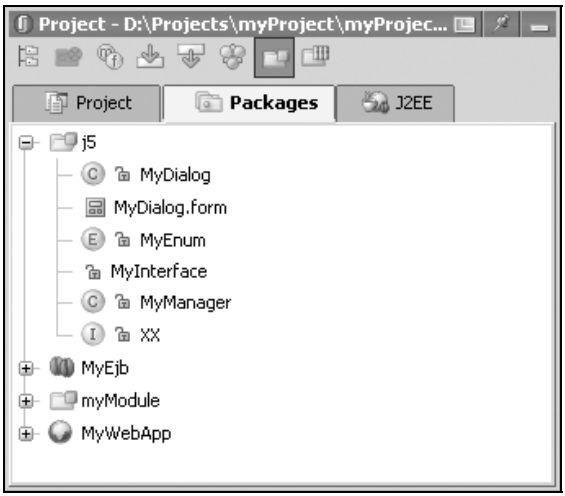


Рис. 3.94. Закладка **Packages** инструментального окна **Project**

Закладка **Packages** (Пакеты) (рис. 3.94) имеет следующие дополнительные команды:

- **Show Modules** (Показывать модули) — команда позволяет включить группировку по пакетам так, как это представлено на рис. 3.94;
- **Show Libraries Contents** (Показывать содержимое библиотек) — команда позволяет включить в представляемую структуру пакетов также информацию о пакетах из библиотек.

С элементами иерархического списка — файлами, классами, интерфейсами и прочими составляющими можно производить различные действия. Список действий задается командами, доступными из контекстного меню, а также из главного меню IntelliJ IDEA.

3.4.8.3. *Commander (Коммандер)*

"Горячие" клавиши: <Alt>+<2>.

Окно **Commander** (рис. 3.95) состоит из двух панелей, на которых отображается файловая структура проекта, одинаковая на левой и правой сторонах. Структура проекта аналогична той, что отображается в инструментальном окне **Project** (Проект) (*см. раздел 3.4.2.2*).

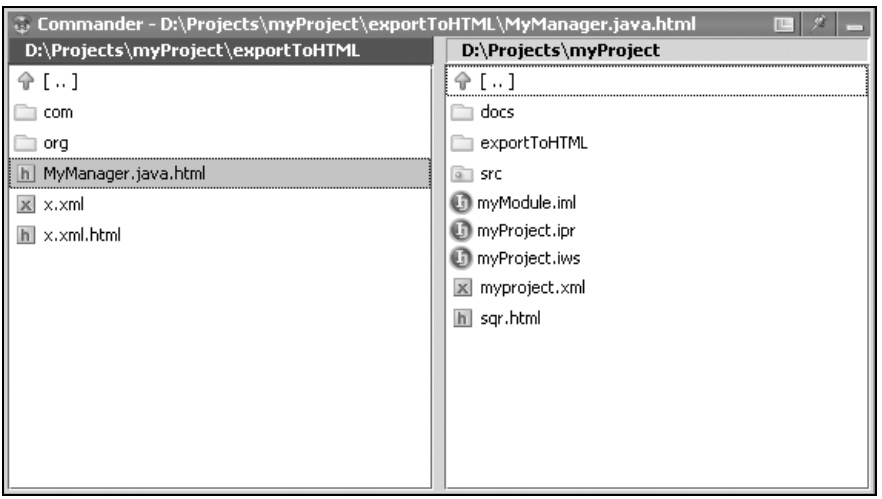


Рис. 3.95. Инструментальное окно **Commander**

Коммандер в IntelliJ IDEA — это некоторый аналог популярного файлового менеджера Norton Commander, также состоящего из двух файловых панелей.

Коммандером очень удобно пользоваться для копирования или переноса файлов, классов, интерфейсов между директориями, пакетами и модулями проекта. Такие операции, по сути, будут запускать соответствующий рефакторинг.

3.4.8.4. *Find (Найти)*

"Горячие" клавиши: <Alt>+<3>.

Инструментальное окно **Find** (Найти) используется для вывода результатов поиска следующими командами:

- **Find Usages** (Поиск использований) — *см. раздел 3.2.3.7*;
- **Find in Path** (Поиск в пути) — *см. раздел 3.2.3.10*;
- **Replace in Path** (Замена в пути) — *см. раздел 3.2.3.11*;
- **Search Structurally** (Структурный поиск) — *см. раздел 6.5*;
- **Replace Structurally** (Структурная замена) — *см. раздел 6.5.*

Общий вид инструментального окна **Find** (Найти) представлен на рис. 3.96.

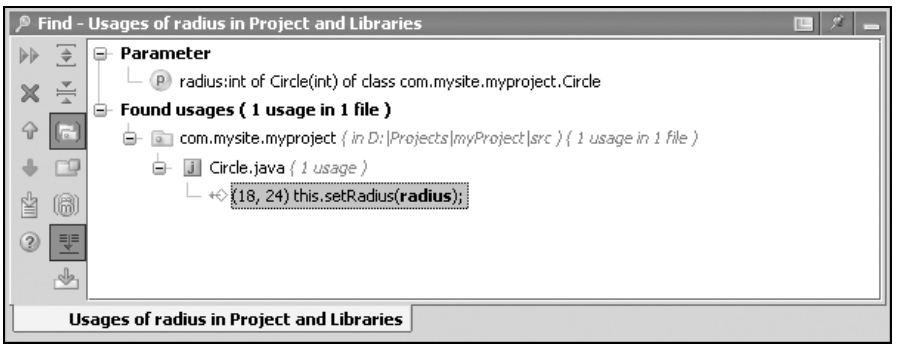


Рис. 3.96. Инструментальное окно **Find**

В левой части окна расположена инструментальная панель (ToolBar) с функциональными кнопками. В зависимости от команды, которая выполняла поиск, набор кнопок может меняться.

Кроме кнопок, персональных для каждого типа поиска, имеются и общие:

- **Rerun search** (Перезапустить поиск) (<Ctrl>+<F5>). Команда перезапускает поиск;
- **Previous/Next Occurrence** (Предыдущий/Следующий найденный фрагмент) (<Ctrl>+<Alt>+<Up>/<Ctrl>+<Alt>+<Down> или кнопки со стрелками <↑> и <↓>) — переход к предыдущему/следующему найденному фрагменту в списке;
- **Export to Text File** (Экспортировать в текстовый файл) — команда экспорта результатов поиска в текстовый файл (рис. 3.97). Для экспорта можно указать дополнительные опции:
  - **Export to file** (Экспортировать в файл) — имя файла для экспорта;
  - **Include usages** (Включить обращения) — опция включает в отчет указания об обращениях — фрагментах найденного текста;

- **Include packages** (Включить пакеты) — опция включает в отчет указания пакетов, в которых были найдены фрагменты;
- **Include methods** (Включить методы) — опция включает в отчет указания методов классов, в которых были найдены фрагменты;
- **Include modules** (Включить модули) — опция включает в отчет информацию о модулях, в которых были найдены фрагменты.

Кнопка **Save** (Сохранить) сохраняет отчет в указанный файл, кнопка **Copy** (Копировать) — копирует его в буфер обмена.

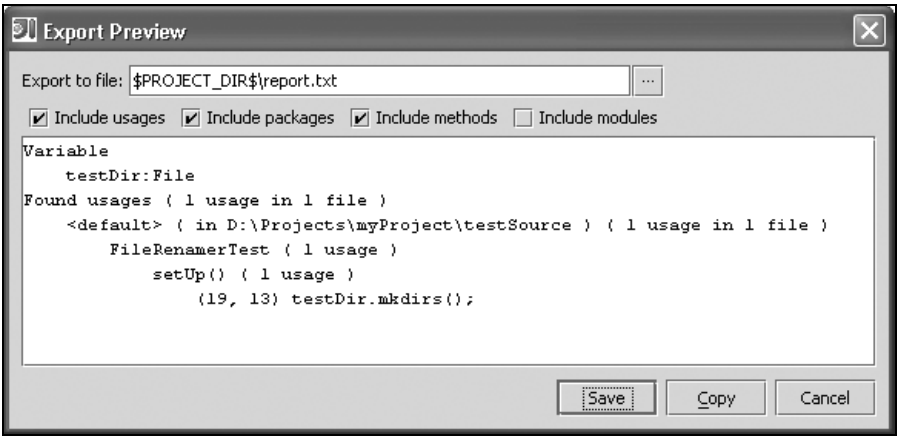


Рис. 3.97. Экспорт результатов поиска в текстовый файл

- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна;
- **Expand/Collapse All** (Развернуть/Свернуть все) (<Ctrl>+<+>/<Ctrl>+<–>) — развернуть/свернуть все группировки найденных фрагментов;
- **Group by packages** (Группировать по пакетам) (<Ctrl>+<P>) — включить группировку найденных фрагментов по пакетам;
- **Group by modules** (Группировка по модулям) (<Ctrl>+<D>) — включить группировку найденных фрагментов по модулям;
- **Merge duplicate lines** (Объединять повторяющиеся строки) — команда позволяет сократить результаты поиска за счет вывода только одной строки на каждый найденный фрагмент, даже если в строке было найдено несколько экземпляров фрагмента;
- **Autoscroll to source** (Перемещаться к исходному тексту) — команда позволяет автоматически открывать в редакторе файл с найденным фрагментом, по мере перемещения по найденной информации в инструментальном окне.

Особенности использования инструментального окна **Find** (Найти) при работе с различными командами описаны в соответствующих разделах.

3.4.8.5. *Run (Запуск)*

"Горячие" клавиши: <Alt>+<4>.

Инструментальное окно **Run** (Запуск) используется в качестве консоли при запуске приложений в IntelliJ IDEA (*см. раздел 9.3*), внешних программ (*см. раздел 4.1.16*), при генерации JavaDoc (*см. раздел 3.2.11.5*), при просмотре визуальной формы пользовательского интерфейса (*см. главу 7*). Внешний вид инструментального окна **Run** (Запуск) представлен на рис. 3.98.

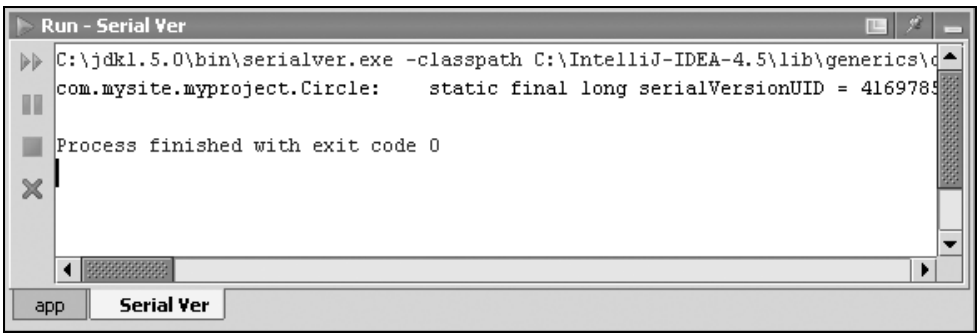


Рис. 3.98. Инструментальное окно **Run**

Данное инструментальное окно в левой части имеет панель инструментов (ToolBar), на которой расположены следующие кнопки:

- **Rerun** (Перезапустить) (<Ctrl>+<F5>) — команда повторного запуска. Она активна, если запущенное приложение уже завершило свою работу;
- **Pause Output** (Приостановить вывод) — команда, позволяющая приостановить печать в консоль. Это своеобразный аналог нажатия кнопки **Pause** (Пауза) в консоли приложения;
- **Stop** (Остановить) (<Ctrl>+<F2>) — команда остановки приложения;
- **Close** (Закрыть) (<Ctrl>+<Shift>+<F4>) — закрыть результаты работы программы. Если программа еще не завершила своей работы, будет выведено предупреждение с опцией принудительного завершения программы (рис. 3.99). Кнопка **Disconnect** (Отсоединить) приведет либо к завершению программы и закрытию ее консоли в инструментальном окне **Run** (Запуск), если включена опция **Terminate the process after disconnect** (Завершить процесс после отсоединения). Либо будет закрыта только консоль приложения, а само приложение останется работать.

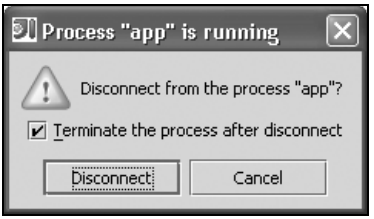


Рис. 3.99. Предупреждение о закрытии консоли работающего приложения

Одновременно может быть запущено неограниченное число приложений. Для каждого из них в инструментальном окне **Run** (Запуск) будет создана отдельная закладка. Например, на рис. 3.98 представлена ситуация, когда запущены два приложения: app и Serial Ver.

3.4.8.6. *Debug (Отладка)*

"Горячие" клавиши: <Alt>+<5>.

Инструментальное окно **Debug** (Отладка) используется IntelliJ IDEA для вывода информации об отладке, а также для управления ходом отладки (рис. 3.100).

Рис. 3.100. Инструментальное окно **Debug**

В левой части окна отладки расположена панель инструментов. Если одновременно отлаживается несколько приложений, инструментальное окно **Debug** (Отладка) дополняется несколькими закладками, расположенными в нижней части окна. Каждая закладка имеет собственный набор вложенных закладок, расположенных в верхней части окна:

- **Console** (Консоль) — на этой закладке расположена консоль отлаживаемого приложения;
- **Threads** (Потоки) — закладка с иерархическим списком потоков, выполняющихся в приложении (рис. 3.101);

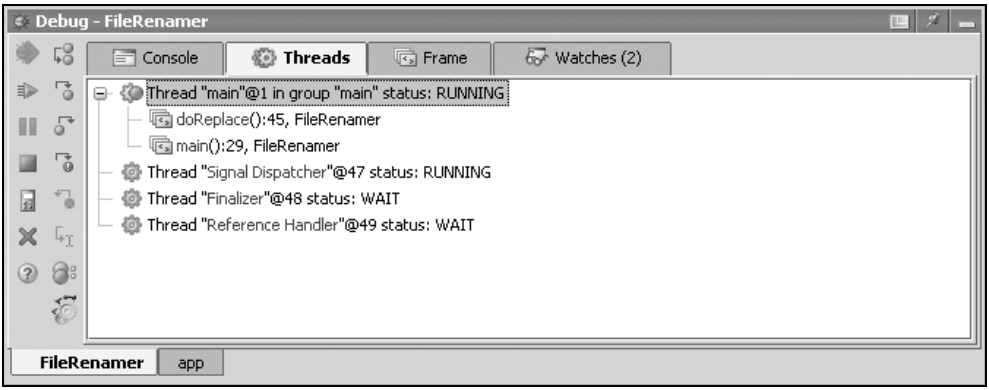


Рис. 3.101. Закладка со списком потоков **Threads** инструментального окна **Debug**

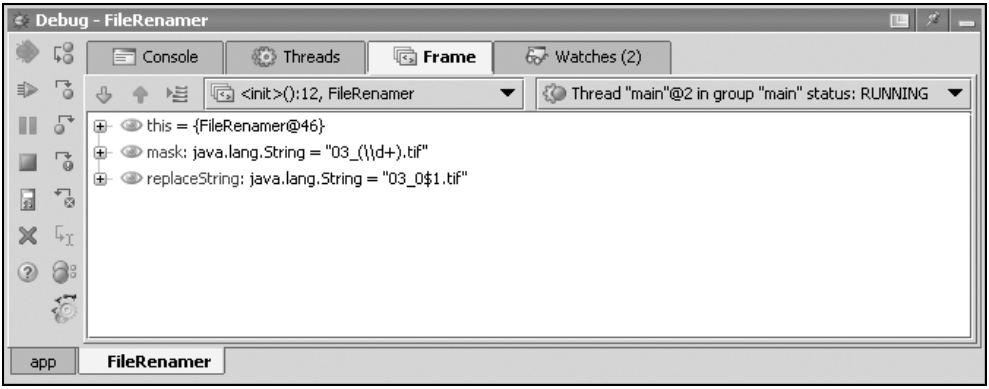


Рис. 3.102. Закладка со списком доступных переменных **Frame** инструментального окна **Debug**

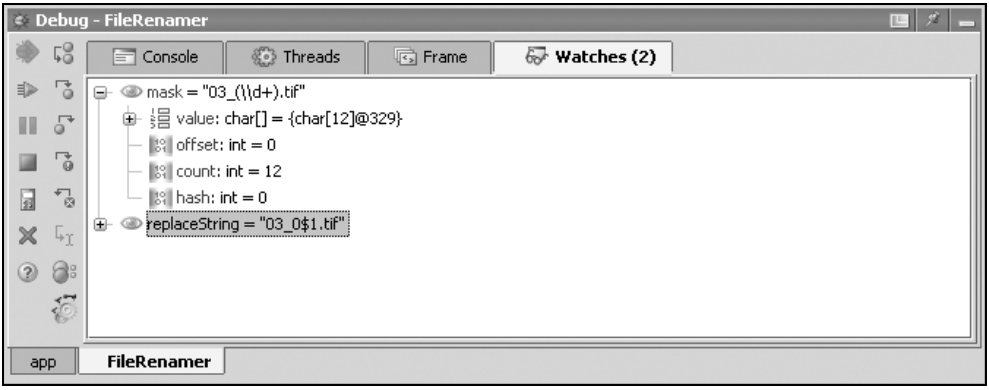


Рис. 3.103. Закладка со списком наблюдаемых переменных **Watches** инструментального окна **Debug**

- **Frame** (Окно стека) — закладка, в которой содержится список переменных, доступных в текущей точке выполнения программы (рис. 3.102);
- **Watches** (Наблюдения) — закладка, в которую можно заносить переменные или выражения, за значением которых вы хотите наблюдать (рис. 3.103).

Панель инструментов данного окна содержит следующие команды:

- **Rerun** (Перезапустить) (<Ctrl>+<F5>) — перезапустить отладку. Данная команда доступна в случае, если отлаживаемая программа завершила свое выполнение;
- **Resume Program** (Возобновить выполнение) (<F9>) — команда продолжает выполнение программы, остановленной в точке останова или с помощью команды **Pause Program** (Приостановить выполнение);
- **Pause Program** (Приостановить выполнение) — команда приостанавливает выполнение отлаживаемой программы;
- **Stop** (Остановить) (<Ctrl>+<F2>) — остановить отладку. Отлаживаемое приложение при этом завершается;
- **Evaluate Expression** (Вычислить выражение) — команда вычисления выражения (*см. раздел 9.4.2*);
- **Close** (Закрыть) (<Ctrl>+<Shift>+<F4>) — команда закрывает окно отладки приложения. Если приложение находится в запущенном состоянии, будет выведено предупреждающее окно (см. рис. 3.99);
- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна;
- **Step Over** (Шаг через) (<F8>) — выполнение программы в пошаговом режиме, не "заходя" внутрь вызовов методов;
- **Step Into** (Шаг внутрь) (<F7>) — пошаговое выполнение с заходом в вызовы методов;
- **Step Out** (Шаг наружу) (<Shift>+<F8>) — пошаговое выполнение с выходом на один уровень выполнения вверх по стеку вызовов методов;
- **Force Step Into** (Принудительный шаг внутрь) (<Shift>+<F7>) — команда работает аналогично команде **Step Into** (Шаг внутрь), однако заходит даже в те методы, в которые обычная команда не вошла бы. Подробнее о настройке таких методов рассказано в *разделе 4.1.9*;
- **Pop Frame** ("Вытолкнуть" окно стека) — команда откатывает содержимое стека, связанное с вызовом некоторого метода. Тем самым выполнение программы возвращается к моменту вызова этого метода;
- **Run to Cursor** (Выполнять до курсора) (<Alt>+<F9>) — выполнение отлаживаемой программы до строки, в которой размещен курсор;
- **View Breakpoints** (Просмотр точек останова) (<Ctrl>+<Sift>+<F8>) — команда отображает диалог со списком точек останова (см. рис. 3.67);

- **Suspend All Threads While Stepping** (Приостановить все потоки во время пошагового выполнения) — команда приостанавливает все потоки, имеющиеся в отлаживаемом приложении, на время пошагового выполнения текущего потока.

Подробнее об отладке приложений рассказано в *разделе 9.4*.

3.4.8.7. *TODO (Заметки)*

"Горячие" клавиши: <Alt>+<6>.

Инструментальное окно **TODO** (Заметки) предназначено для отображения списка комментариев к программному коду специфической формы (рис. 3.104).

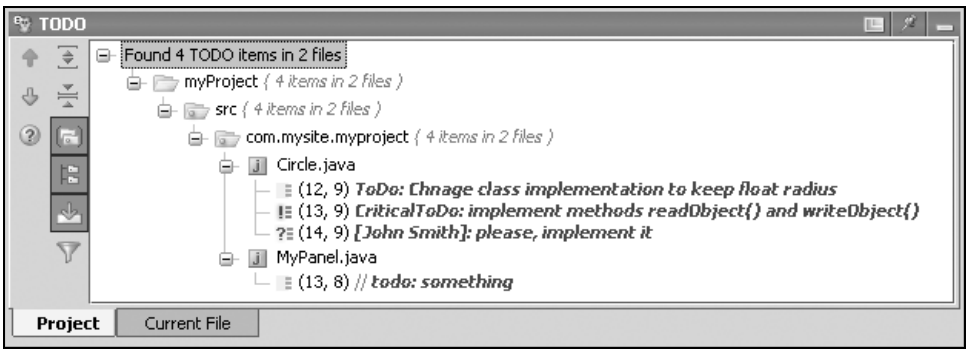


Рис. 3.104. Инструментальное окно **TODO**

В левой части окна расположена панель инструментов со следующими кнопками:

- **Previous/Next TODO** (Предыдущая/Следующая заметка) (<Ctrl>+<Alt>+<Up>/<Ctrl>+<Alt>+<Down> или кнопки со стрелками <↑> и <↓>) — переход к предыдущему/следующему пункту в списке дел;
- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна;
- **Expand/Collapse All** (Развернуть/Свернуть все) (<Ctrl>+<+>/<Ctrl>+<−>) — развернуть/свернуть все группировки в списке заметок;
- **Group By Packages** (Группировать по пакетам) (<Ctrl>+<P>) — группировать заметки по пакетам, к которым принадлежат файлы с соответствующими комментариями;
- **Flatten Packages** (Плоские пакеты) — команда переключает режим отображения пакетов. Когда она выключена, пакеты отображаются в виде иерархического списка, например, com/mysite/myproject. Когда выключена — в виде плоского списка с полным именем, например, com.mysite.myproject;

- **Autoscroll to Source** (Переход к исходному тексту) — команда, позволяющая автоматически выводить в окне редактора содержимое файлов, к которым относится тот или иной комментарий из списка заметок.

Подробнее о настройке окна просмотра заметок можно прочитать в *разделе 4.1.18*.

3.4.8.8. *Structure (Структура)*

"Горячие" клавиши: <Alt>+<7>.

Инструментальное окно **Structure** (Структура) предназначено для просмотра структуры редактируемого файла. Отображение структуры поддерживается для любых Java- и XML-файлов.

Общий вид окна **Structure** (Структура) для Java-файла показан на рис. 3.105, для XML-файла — на рис. 3.106.

В верхней части окна расположена панель инструментов. При просмотре структуры Java-файла доступны следующие кнопки:

- **Sort Alphabetically** (Сортировать по алфавиту) — команда включает сортировку по алфавиту для содержимого структуры файла;
- **Sort by Visibility** (Сортировать по области видимости) — команда включает сортировку полей и методов класса на основе их модификаторов доступа;
- **Group Methods by Defining Type** (Группировать методы по определяющему типу) — команда включает дополнительную группировку методов в зависимости от того, являются ли они имплементацией или переопределением методов из интерфейсов или базового класса;
- **Show properties** (Показывать свойства) — команда включает отображение полей класса, для которых определены методы `get` и/или `set`;

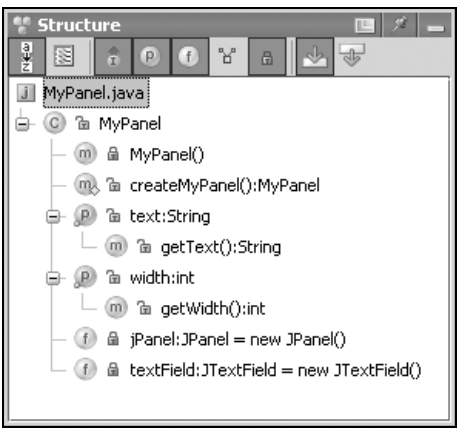


Рис. 3.105. Инструментальное окно **Structure** для Java-файла

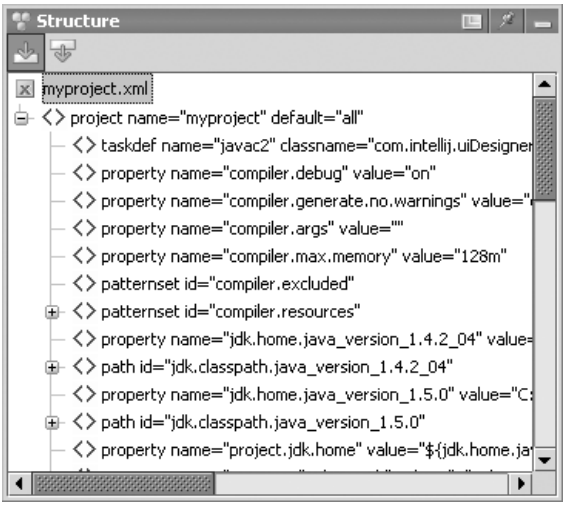


Рис. 3.106. Инструментальное окно **Structure** для XML-файла

- **Show fields** (Показывать поля) — команда включает отображение прочих полей класса;
- **Show Inherited Members** (Показывать наследуемых членов класса) — команда включает отображение полей и методов, которые класс наследует от своего базового класса;
- **Show Non-public Members** (Показывать не только `public`-члены класса) — команда включает отображение всех полей и методов класса, в том числе и тех, область видимости которых объявлена как `private`, `protected` и `package`.

Следующие две команды в панели инструментов доступны для структуры любых файлов:

- **Autoscroll to Source** — команда, позволяющая автоматически прокручивать содержимое окна редактора вслед за перемещением по структуре файла в инструментальном окне;
- **Autoscroll from Source** — команда, обратная предыдущей. Позволяет автоматически перемещать содержимое инструментального окна в соответствии с положением курсора в редакторе.

3.4.8.9. *Hierarchy (Иерархия)*

"Горячие" клавиши: <Alt>+<8>.

Инструментальное окно **Hierarchy** (Иерархия) используется для отображения результатов работы трех команд:

- **Type Hierarchy** (Иерархия типов) (<Ctrl>+<H>) — *см. раздел 3.2.4.16*;

- **Method Hierarchy** (Иерархия методов) (<Ctrl>+<Shift>+<H>) — *см. раздел 3.2.4.17*;
- **Call Hierarchy** (Иерархия вызовов) (<Ctrl>+<Alt>+<H>) — *см. раздел 3.2.4.18.*

Общий вид окна в зависимости от результатов представлен на рис. 3.107—3.109.

В верхней части окна располагается панель инструментов, набор кнопок в которой различается в зависимости от того, какая команда использует окно **Hierarchy** (Иерархия) в настоящий момент.

Для команды **Type Hierarchy** (Иерархия типов) набор кнопок слева дополняется тремя кнопками, работающими в режиме переключателей (см. рис. 3.107):

- **Class Hierarchy** (Иерархия классов) — отображать иерархию класса, иерархия которого просматривается в данный момент;
- **Supertypes Hierarchy** (Иерархия базового типа) — отображать иерархию для базового типа просматриваемого класса;
- **Subtypes Hierarchy** (Иерархия производных типов) — отображать иерархию производных типов просматриваемого класса.

Для команды **Method Hierarchy** (Иерархия методов) имеется одна собственная кнопка на панели инструментов **Hide classes where the method legally is not implemented** — скрывать классы в иерархии вызова метода, в которых исследуемый метод не переопределяется на самом деле.

В случае просмотра иерархии вызова метода в нижней части инструментального окна дополнительно располагается справочная информация о значках, используемых для отображения информации о методах в иерархии:

- **+ method is defined in the class** — метод определен в классе;
- **− method is not defined in the class but defined in superclass** — метод не определен в классе, но определен в базовом классе;
- **! method should be defined since the class is not abstract** — метод должен быть определен в классе, поскольку класс не имеет модификатора `abstract`.

В режиме отображения информации об иерархии вызова метода (**Call Hierarchy**) (см. рис. 3.109) панель инструментов окна **Hierarchy** (Иерархия) дополняется двумя переключателями, а также опцией выбора области вызова:

- **Caller Methods Hierarchy** (Иерархия вызывающего метода) — команда отображает иерархию вызова методов от просматриваемого метода до точки его вызова;
- **Callee Methods Hierarchy** (Иерархия вызываемого метода) — команда отображает иерархию вызовов методов, производимых просматриваемым методом;

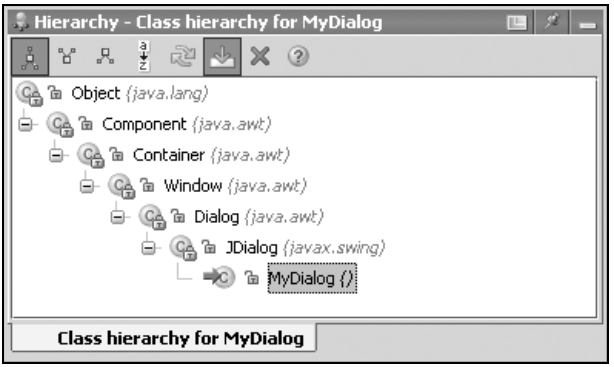


Рис. 3.107. Инструментальное окно **Hierarchy** с результатами работы команды **Type Hierarchy**

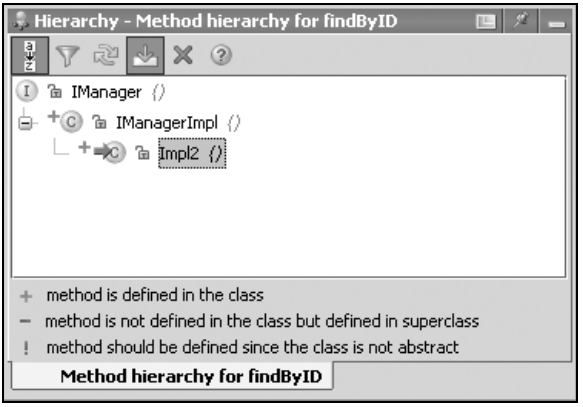


Рис. 3.108. Инструментальное окно **Hierarchy** с результатами работы команды **Method Hierarchy**

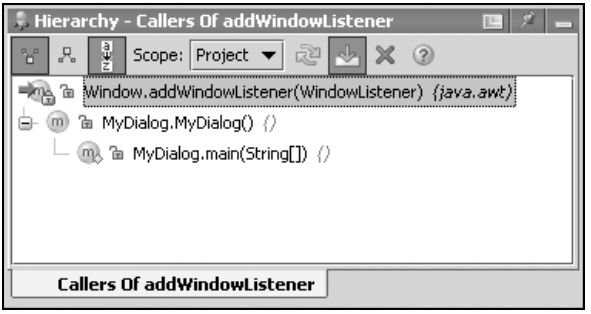


Рис. 3.109. Инструментальное окно **Hierarchy** с результатами работы команды **Call Hierarchy**

- **Scope** (Область просмотра) — опция задает область просмотра иерархии вызовов метода:
  - **Project** (Проект) — вызовы метода из файлов проекта;
  - **All** (Все) — вызовы метода из файлов проекта, а также из всех библиотек;
  - **This Class** (Текущий класс) — вызовы метода в пределах текущего класса.

Также на панели инструментов имеются следующие общие кнопки:

- **Sort Alphabetically** (Сортировать по алфавиту) — команда сортировки содержимого окна **Hierarchy** (Иерархия) по алфавиту;
- **Refresh** (Обновить) (<Ctrl>+<F5>) — обновить содержимое окна иерархии;
- **Autoscroll to Source** — команда, позволяющая автоматически прокручивать содержимое окна редактора вслед за перемещением по иерархии в инструментальном окне;
- **Close** (Закрыть) (<Ctrl>+<Shift>+<F4>) — закрыть окно;
- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна.

3.4.8.10. *Ant Build (Сборка Ant)*

Инструментальное окно файлов сборки утилиты Ant представлено на рис. 3.110.

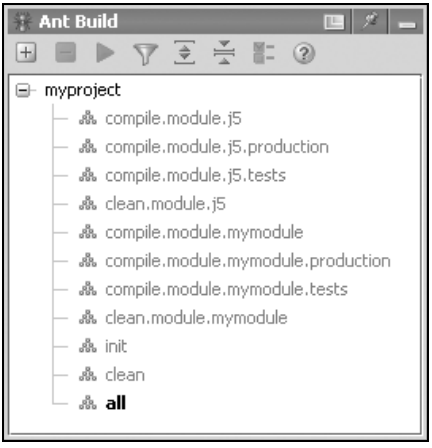


Рис. 3.110. Инструментальное окно **Ant Build**

Инструментальное окно **Ant Build** (Сборка Ant) содержит список файлов сборки Ant с содержащимися в них целями сборки. В верхней части окна

Ant Build (Сборка Ant) содержится панель инструментов со следующими кнопками:

- **Add** (Добавить) — добавить файл сборки Ant в список;
- **Remove** (Удалить) — удалить файл сборки Ant из списка;
- **Run** (Запустить) — запустить команду из файла сборки Ant;
- **Filter targets** (Фильтровать цели) — данная команда позволяет отфильтровать цели сборки, представленные в списке соответственно настройкам (*см. главу 13*);
- **Expand/Collapse All** (Развернуть/Свернуть все) (<Ctrl>+<+>/<Ctrl>+<−>) — развернуть/свернуть все группировки в окне;
- **Properties** (Свойства) — команда редактирования свойств выделенного файла сборки Ant;
- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна.

Подробнее о работе с утилитой Ant в IntelliJ IDEA рассказано в *главе 13*.

3.4.8.11. *CVS*

Данное инструментальное окно используется для отображения результатов работы команд системы контроля версий CVS. Подробнее об использовании данного инструментального окна рассказано в *разделе 12.3*.

3.4.8.12. *Dependency Viewer (Просмотр зависимостей)*

Инструментальное окно **Dependency Viewer** (Просмотр зависимостей) используется для отображения результатов работы команды **Analyze Dependencies** (Анализ зависимостей) (*см. раздел 6.4.2*). Внешний вид окна представлен на рис. 3.111.

Окно разделено на три части. В верхней части расположены два окна, между содержимым которых производится анализ зависимостей. В левом окне выбирается пакет или класс, в правой части выбираются другие возможные пакеты, и между ними производится анализ зависимостей, результат которого выводится в нижнем окне.

В верхней части окна расположена панель инструментов, команды которой воздействуют на два верхних окна со списками:

- **Close** (Закрыть) — команда закрывает инструментальное окно;
- **Rerun** (Перезапустить) (<Ctrl>+<F5>) — команда перезапуска поиска зависимостей;
- **Flatten Packages** (Плоские пакеты) — команда переключает режим отображения пакетов. Когда она выключена, пакеты отображаются в виде иерархического списка, например: com/mysite/myproject. Когда выключена — в виде плоского списка с полным именем, например: com.mysite.myproject;

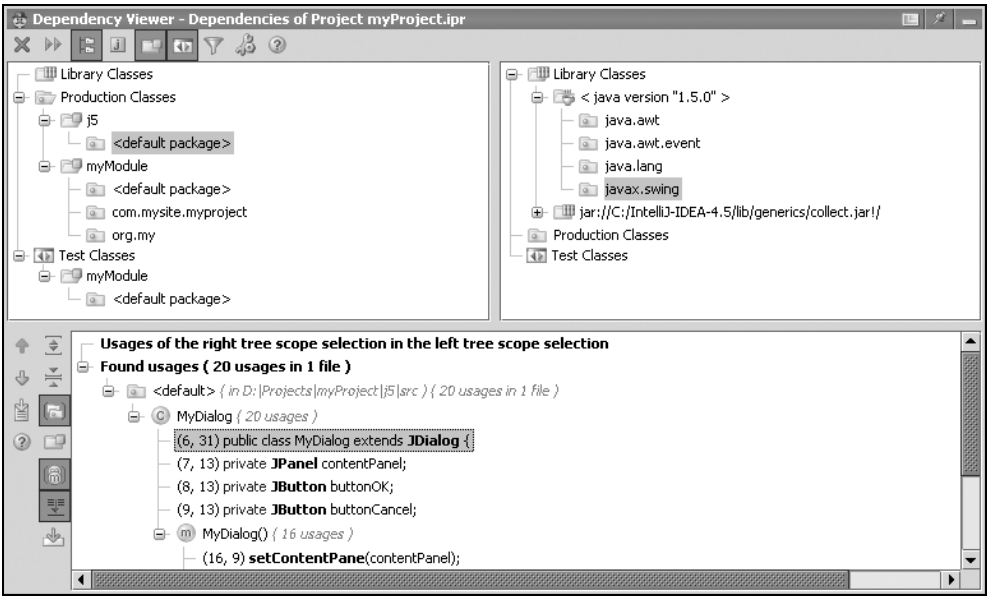


Рис. 3.111. Инструментальное окно **Dependency Viewer**

- **Show Files** (Показывать файлы) — команда включает отображение файлов внутри пакетов;
- **Show Modules** (Показывать модули) — команда включает отображение модулей;
- **Group by Scope Type** (Группировать по типу) — команда включает дополнительную группировку элементов анализа (пакетов и классов) по типу, например, тесты, обычный код и т. п.;
- **Show Illegals Only** (Показывать только нарушения) — команда включает отображение только тех пакетов или классов, которые нарушают правила зависимостей для проекта (*см. раздел 6.4.2*);
- **Edit Rules** (Редактировать правила) — команда, открывающая диалог настройки зависимостей для проекта (*см. раздел 6.4.2*);
- **Help** (Помощь) (<F1>) — вывод контекстной справочной информации о работе данного инструментального окна.

Нижняя часть окна имеет собственную панель инструментов:

- **Previous/Next Occurrence** (Предыдущий/Следующий пункт в списке) (<Ctrl>+<Alt>+<Up>/<Ctrl>+<Alt>+<Down> или кнопки со стрелками <↑> и <↓>) — переход к предыдущему/следующему пункту в списке найденных зависимостей;
- **Export to Text File** (Экспорт в текстовый файл) — команда экспорта результатов анализа зависимостей в текстовый файл (см. рис. 3.97). Команда

аналогична команде экспорта в текстовый файл для результатов поиска (*см. раздел 3.4.8.4*);

- **Expand/Collapse All** (Развернуть/Свернуть все) (<Ctrl>+<+>/<Ctrl>+<–>) — развернуть/свернуть все группировки в окне;
- **Group by Packages** (Группировать по пакетам) (<Ctrl>+<P>) — команда группирует найденные зависимости по пакетам;
- **Group by Modules** (Группировать по модулям) (<Ctrl>+<D>) — команда группирует найденные зависимости по модулям;
- **Group by Methods** (Группировать по методам) (<Ctrl>+<M>) — команда группирует найденные зависимости по методам;
- **Merge duplicate lines** (Объединять повторяющиеся строки) — команда сокращает список найденных зависимостей, выводя зависимости из одной строки в качестве одного элемента списка;
- **Autoscroll to Source** (Прокрутка к исходному тексту) — команда, позволяющая автоматически прокручивать содержимое окна редактора вслед за перемещением по найденным зависимостям в инструментальном окне.

Подробнее поиск зависимостей рассмотрен в *разделе 6.4.2*.

3.4.8.13. *Inspection (Инспекции)*

Инструментальное окно **Inspection** (Инспекции) предназначено для отображения результатов инспекции кода (рис. 3.112).

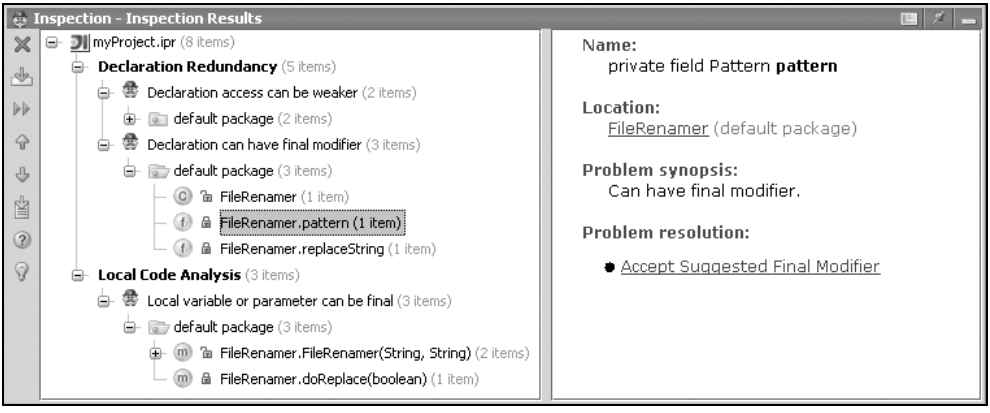


Рис. 3.112. Инструментальное окно **Inspection**

Окно состоит из двух частей: в левой части отображается иерархический список инспекций, что-либо обнаруживших в коде. В правой части содержится описание конкретного пункта инспекции.

В левой части окна расположена панель инструментов со следующими кнопками:

- **Close** (Закрыть) — команда закрывает инструментальное окно;
- **Autoscroll to Source** — команда, позволяющая автоматически прокручивать содержимое окна редактора вслед за перемещением по результатам инспекций;
- **Rerun Inspection** (Перезапустить инспекции) (<Ctrl>+<F5>) — команда повторно запускает инспекции;
- **Go Prev/Next Problem** (Предыдущая/Следующая проблема) (<Ctrl>+<Alt>+<Up>/<Ctrl>+<Alt>+<Down> или кнопки со стрелками <↑> и <↓>) — переход к предыдущему/следующему пункту в списке найденных инспекциями проблем в коде;
- **Export to HTML** (Экспорт в HTML) — команда экспорта результатов инспекций в HTML. Данная команда открывает диалог, в котором предлагает ввести директорию для экспорта HTML-файлов (**Output directory**) (рис. 3.113). Также имеется опция **Open generated HTML in browser** — открыть сгенерированный HTML-файл в браузере;

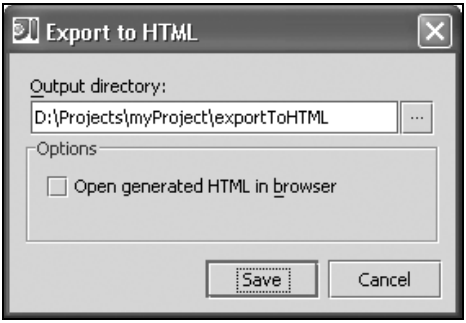


Рис. 3.113. Экспорт результатов инспекций в HTML-файл

- **Help** (Помощь) (F1) — вывод контекстной справочной информации о работе данного инструментального окна;
- **Apply a quickfix** (Применить "подсказку") (<Alt>+<Enter>) — команда применения "подсказки". Она позволяет автоматически внести предлагаемое исправление в код.

Подробнее о "подсказках" можно прочитать в *разделе 6.1.*

3.5. Панель статуса (Status Bar)

Панель статуса IntelliJ IDEA имеет несколько разделов (рис. 3.114).

Рис. 3.114. Общий вид панели статуса IntelliJ IDEA

Панель статуса имеет следующие разделы:

- левая самая широкая часть используется для вывода различных сообщений;
- далее следует раздел, отображающий текущую позицию курсора в окне редактора. На рис. 3.114 курсор находится в десятой строке, четырнадцатом столбце (10:14);
- следующий короткий раздел используется для отображения пиктограммы "замочек" для файлов, имеющих доступ только на чтение. Если на этой пиктограмме сделать двойной щелчок мышью, можно изменить режим доступа к файлу;
- далее следует раздел, в котором отображается текущий режим ввода. Значений может быть два:
  - **Insert** (Вставка);
  - **Overwrite** (Замена) — в этом случае курсор в редакторе приобретает вид блока;
- следующий раздел содержит переключатель **Import Popup** (Всплывающее предложение импорта), имеющий два значения:
  - **ON** — IntelliJ IDEA будет автоматически предлагать вставить строку импорта для класса (рис. 3.115);
  - **OFF** — автоматического предложения вставить импорт не будет;

Рис. 3.115. Всплывающее предложение импорта

- в последнем разделе находится индикатор использованной памяти с кнопкой принудительного запуска системы сборки мусора (Garbage Collection). На этом индикаторе может меняться как количество использованной памяти, так и общее значение памяти. Это не ошибка ☺. Дело в том, что программе на Java (а IntelliJ IDEA — это чистое Java-приложение) заранее не известно, сколько памяти ей доступно — это оп-

ределяется виртуальной машиной. Виртуальная машина может лишь сообщить Java-программе, сколько памяти из некоторого объема она использовала. Именно эти два значения и отображаются в правом нижнем углу окна IntelliJ IDEA.

3.6. Окна редактирования

Окна для редактирования файлов занимают большую часть окна IntelliJ IDEA (рис. 3.116). Одновременно может быть открыто несколько окон редактора.

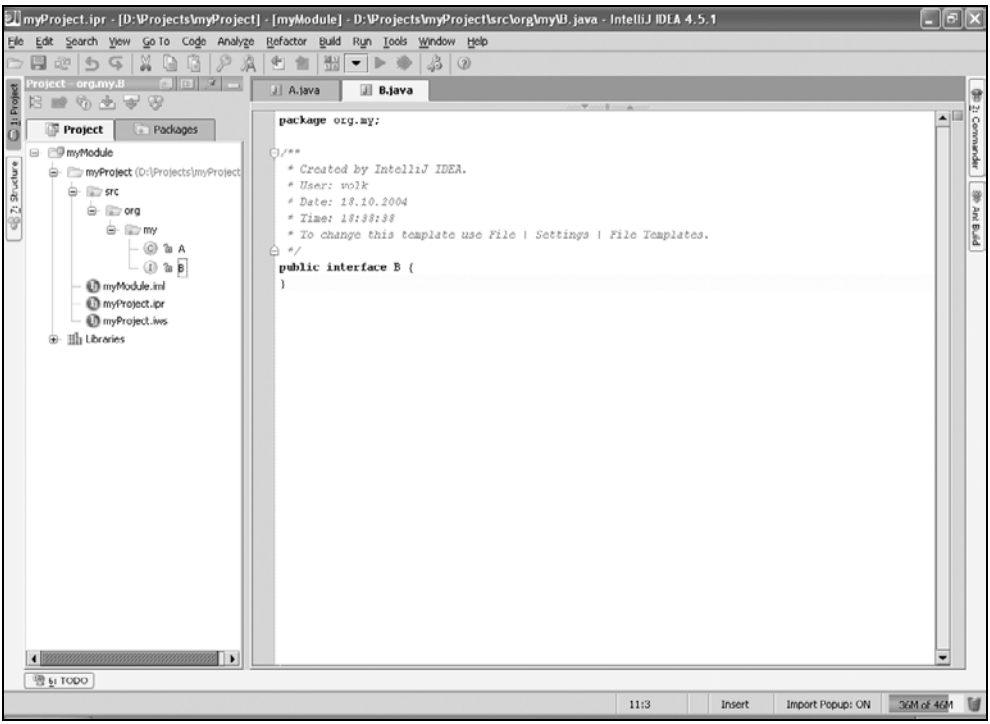


Рис. 3.116. Общий вид окна проекта в IntelliJ IDEA с двумя файлами, открытыми в редакторе

3.6.1. Закладки редакторов

Каждое окно редактора размещается на отдельной закладке — таким образом, между окнами редактора очень удобно перемещаться. Для перехода на тот или иной файл достаточно переключиться на соответствующую закладку.

Максимальное число закладок ограничено и может быть настроено с помощью диалога настроек редактора (*см. раздел 4.1.3*).

3.6.1.1. Разделение закладок редактора

Если требуется одновременно просматривать несколько файлов, можно разделить закладки на две и более части. Для разделения закладок используются специальные команды из меню **File** (Файл), которые также продублированы в контекстном меню каждой закладки:

- **Split Tab Group Vertically** (Разделить группу закладок вертикально). Команда создает две панели (или две группы закладок), разделенных по вертикали, на одной из которых располагаются все закладки редакторов. В противоположной группе закладок находится то окно редактора, на котором была выполнена данная команда (рис. 3.117);

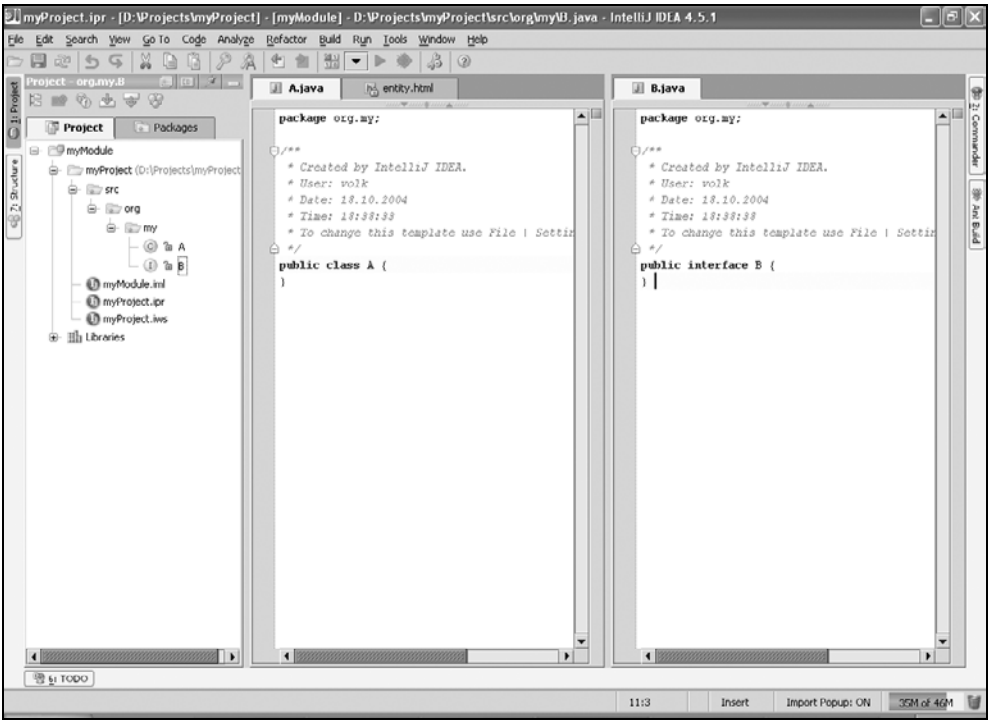


Рис. 3.117. Вертикальное разделение группы закладок с окнами редактора

- **Split Tab Group Horizontally** (Разделить группу закладок горизонтально). Команда, аналогичная предыдущей, за исключением того, что группы закладок будут разделены горизонтально;
- **Move to Opposite Tab Group** (Переместить в противоположную группу закладок). Команда перемещает текущую закладку в противоположную группу;

- **Change Tab Group Orientation** (Изменить ориентацию группы закладок). Команда изменяет разделение групп закладок с вертикальной на горизонтальную и наоборот.

3.6.1.2. Фиксация закладки

По мере открытия файлов в редакторе IntelliJ IDEA убирает закладки с файлами, которые редактировались последними. Однако есть возможность закрепить закладки так, чтобы они не убирались автоматически по мере открытия новых закладок с редакторами.

Для фиксации закладки служит команда **Pin Active Tab** (Закрепить активную закладку) из раздела меню **Window** (Окно) главного меню IntelliJ IDEA. Также команда для закрепления закладки продублирована в контекстном меню закладки.

Закрепленная закладка отображается с небольшим значком кнопки, показывающим, что закладка закреплена (рис. 3.118).

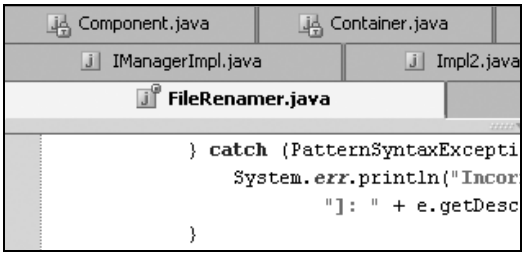


Рис. 3.118. Зафиксированная закладка редактора

3.6.2. Поля редактора

Окна редактора имеют левое и правое поле, на которых может отображаться дополнительная информация о содержимом файла (рис. 3.119).

3.6.2.1. Правое поле

На правом поле отображаются цветные линии, нажимая на них левой кнопкой мыши можно переходить на места в коде, которые этим линиям соответствуют. Это своего рода закладки. Если подвести курсор мыши к закладке, отобразится всплывающая подсказка с сообщением о том, на что указывает закладка.

Закладки бывают:

- красного цвета с указанием на ошибку в коде;
- желтого цвета с указанием на замечание к коду;

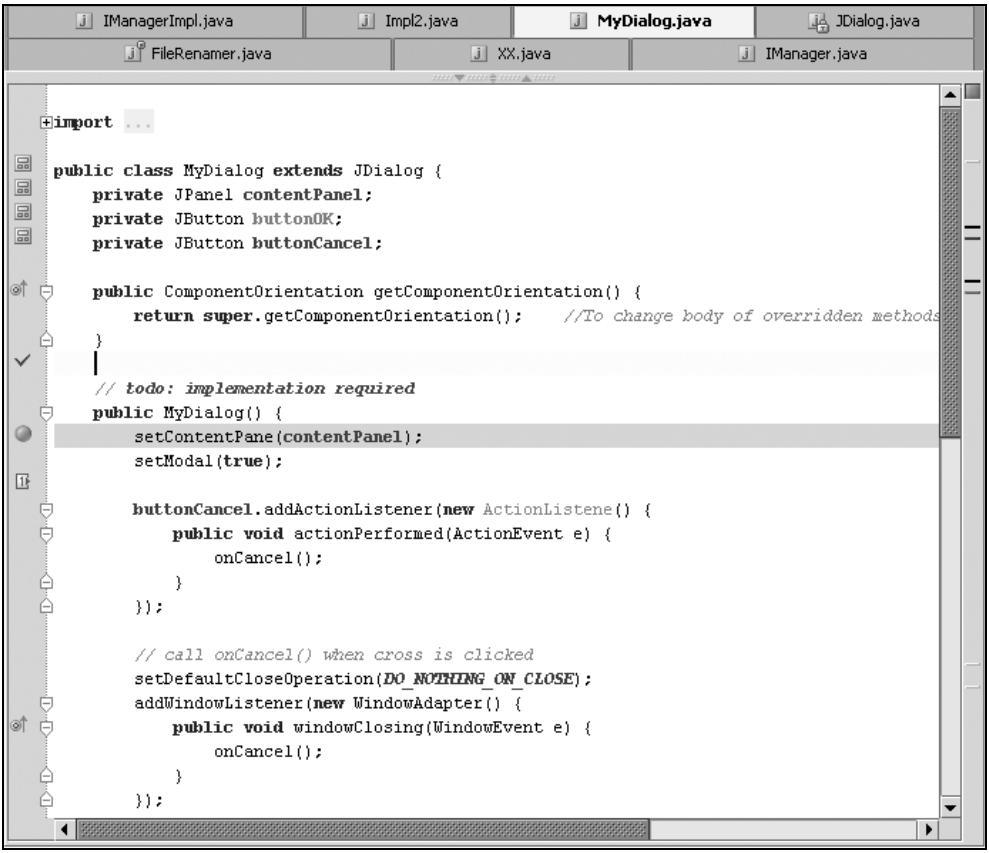


Рис. 3.119. Пометки на полях редактора

- синего цвета с указанием на пункт из списка заметок (TODO), (*см. разделы 3.4.8.7, 4.1.18*);
- черного цвета с указанием места установки обычной или цифровой закладки (*см. разделы 3.2.2.12, 3.2.2.13*).

В самом верху правого поля располагается квадратный индикатор — индикатор анализа текста, который может принимать четыре цветовых значения:

- серый — анализ текста не начался;
- красный — в коде файла есть ошибки;
- желтый — к коду файла есть замечания;
- зеленый — с кодом все в порядке.

Если в индикаторе отображается значок глаза, значит, анализ еще не завершен (рис. 3.120).

Рис. 3.120. Индикатор анализа текста

3.6.2.2. Левое поле

В левом поле отображаются значки (см. рис. 3.119):

- связей класса и полей класса с формой пользовательского интерфейса (*см. главу 7*);
- значки простых и цифровых закладок (*см. раздел 3.2.2.12*);
- значки точек останова (*см. раздел 9.4*);
- значки имплементации и переопределения методов класса;
- указатели блока кода (рис. 3.121). С помощью такой "скобки" IntelliJ IDEA указывает на то, где блок начинается и где заканчивается.

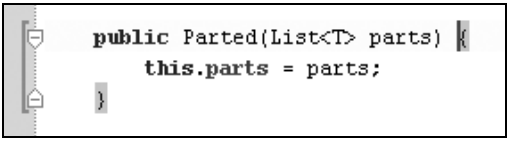


Рис. 3.121. Выделение блока кода

Значки на левой панели также имеют собственную функциональность — на них можно нажимать мышью для перехода к тому файлу, классу, методу или позиции в файле, к которому они относятся.

Кроме значков на левой панели располагаются метки для представления содержимого документа в сокращенном или развернутом виде (*см. раздел 3.2.4.19*).

На левой панели также может отображаться нумерация строк (рис. 3.122).

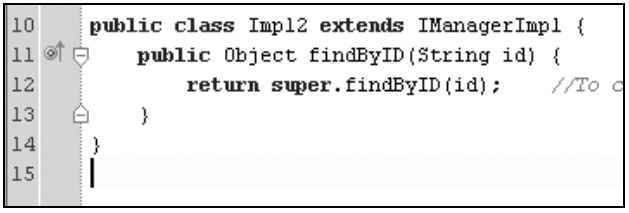


Рис. 3.122. Нумерация строк в левом поле редактора

3.6.3. Дополнительные визуальные элементы

Различные визуальные вспомогательные эффекты возможны также в самом окне редактора. К ним можно отнести линию-ограничитель правого поля, разделители методов (рис. 3.123) (*см. раздел 4.1.3*).

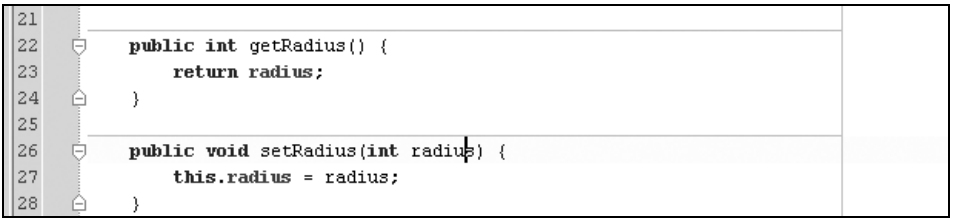


Рис. 3.123. Разделители методов и правая граница

3.6.4. Команды редактора

Все действия, которые можно сделать в редакторе, задаются командами. Команд очень много — от простых команд перемещения курсора до сложных, работающих в паре с другой функциональностью IntelliJ IDEA, например, выбор параметра при работе "живых шаблонов" (*см. раздел 4.1.13*).

3.6.4.1. Настройка команд редактора

IntelliJ IDEA обладает очень гибкими настройками редактора, с помощью которых можно настроить практически любые команды, которые можно выполнять при редактировании. Можно даже переназначить клавиши перемещения курсора, чтобы клавиша перемещения курсора влево вызывала его перемещение вправо и т. п.

Такая гибкая настройка позволяет каждому программисту настроить персональные сочетания клавиш так, как ему это удобно. Если кто-то всю жизнь работал в Emacs под Unix, с помощью специальной схемы клавиш для Emacs он сможет использовать в IntelliJ IDEA те же самые команды, что и в Emacs.

С помощью настройки команд редактора в IntelliJ IDEA можно проэмулировать практически любые среды разработки и редактирования текста. Некоторые сложности возникают только с полной эмуляцией так называемых командных редакторов наподобие Vi. В таких редакторах все команды, в том числе и те, что в IntelliJ IDEA производятся с помощью дополнительных элементов пользовательского интерфейса — диалогов и вспомогательных окон, выполняются в виде ввода определенной последовательности сочетания символов. Тем не менее, основные команды работы с текстом можно настроить даже для таких редакторов. Хотя, если честно, не очень

понятно зачем, поскольку сочетания клавиш IntelliJ IDEA по умолчанию весьма удобны, стоит лишь поработать в IntelliJ IDEA пару дней.

Для истинных поклонников Vi можно предложить использовать специальный плагин, полностью эмулирующий поведение Vi в IntelliJ IDEA (*см. главу 15*).

Практически все команды редактора вводятся с помощью клавиатуры в виде нажатия отдельных клавиш или как сочетание. Сочетания клавиш редактора можно настроить с помощью диалога настройки сочетаний клавиш (*см. раздел 4.1.15*).

3.6.4.2. Полный перечень команд редактора

Далее мы приведем полный перечень команд редактора. Для тех команд, смысл которых очевиден, например, переместить курсор на одну позицию влево (даже эту команду можно отдельно настроить), мы не будет давать пояснений. Пояснения будут приведены для более сложных команд. В скобках приведенные сочетания клавиш активации команд в схеме сочетаний клавиш по умолчанию (*см. раздел 4.1.15*).

- **Backspace** (<Backspace>, <Shift>+<Backspace>) — удалить символ слева от курсора.
- **Choose Lookup Item** (<Enter>) — выбор пункта поиска. Данная команда работает в случае выбора одного из предложенных вариантов завершения кода. Выбранный пункт будет вставлен в месте вызова команды в коде. Подробнее об этой функциональности можно прочитать в *разделе 6.1.5*.
- **Choose Lookup Item Replace** (<Tab>) — выбор пункта поиска для замены. Данная команда работает в случае выбора одного из предложенных вариантов завершения кода. Выбранный пункт будет вставлен взамен имеющейся кодовой конструкции. Подробнее об этой функциональности можно прочитать в *разделе 6.1.5*.
- **Move Caret To Code Block End** (<Ctrl>+<]>) — переместить курсор к концу блока кода.
- **Move Caret To Code Block End with Selection** (<Ctrl>+<Shift>+<]>) — переместить курсор к концу блока кода вместе с выделением.
- **Move Caret To Code Block Start** (<Ctrl>+<[>) — переместить курсор к началу блока кода.
- **Move Caret To Code Block Start with Selection** (<Ctrl>+<Shift>+<[>) — переместить курсор к началу блока кода вместе с выделением.
- **Complete Current Statement** (<Ctrl>+<Shift>+<Enter>) — завершить текущий оператор. Данная команда распознает текущий контекст вызова и может завершить незаконченную конструкцию, например, как это показано на рис. 3.124.

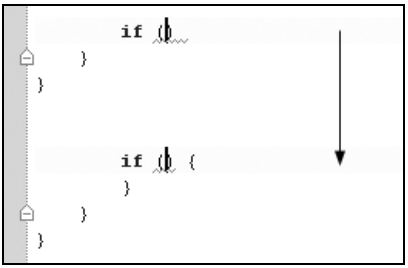


Рис. 3.124. Вариант работы команды **Complete Current Statement**

- **Cut up to Line End** — удалить до конца строки.
- **Delete Line at Caret** (<Ctrl>+<Y>) — удалить строку с курсором.
- **Delete to Word End** (<Ctrl>+<Delete>) — удалить до конца слова.
- **Delete to Word Start** (<Ctrl>+<Backspace>) — удалить до начала слова.
- **Down** (<Down>) — курсор вниз.
- **Down with Selection** (<Shift>+<Down>) — курсор вниз вместе с выделением.
- **Duplicate Line or Block** (<Ctrl>+<D>) — повторить строку или выделенный блок.
- **Enter** (<Enter>) — новая строка.
- **Escape** (<Escape>) — команда отмены.
- **Indent Selection** (<Tab>) — установить отступ красной строки для выделенного блока.
- **Join Lines** (<Ctrl>+<Shift>+<J>) — объединить строки.
- **Left** (Left) — курсор влево.
- **Left with Selection** (<Shift>+<Left>) — курсор влево вместе с выделением.
- **Move Caret to Line End** (<End>) — курсор в конец строки.
- **Move Caret to Line End with Selection** (<Shift>+<End>) — курсор в конец строки вместе с выделением.
- **Move Caret to Line Start** (<Home>) — курсор в начало строки.
- **Move Caret to Line Start with Selection** (<Shift>+<Home>) — курсор в начало строки вместе с выделением.
- **Move Down and Scroll** — курсор на позицию вниз вместе с перемещением окна редактора.
- **Move Down and Scroll with Selection** — курсор на позицию вниз вместе с перемещением окна редактора и вместе с выделением.
- **Move Caret to Page Bottom** (<Ctrl>+<Page Down>) — курсор в конец страницы.

- **Move Caret to Page Bottom with Selection** (<Ctrl>+<Shift>+<Page Down>) — курсор в конец страницы вместе с выделением.
- **Move Caret to Page Top** (<Ctrl>+<Page Up>) — курсор в начало страницы.
- **Move Caret to Page Top with Selection** (<Ctrl>+<Shift>+<Page Up>) — курсор в начало страницы вместе с выделением.
- **Move Up and Scroll** — курсор на позицию вверх вместе с перемещением окна редактора.
- **Move Up and Scroll with Selection** — курсор на позицию вверх вместе с перемещением окна редактора и вместе с выделением.
- **Move Caret to Next Word** (<Ctrl>+<Right>) — курсор к следующему слову.
- **Move Caret to Next Word with Selection** (<Ctrl>+<Shift>+<Right>) — курсор к следующему слову вместе с выделением.
- **Page Down** (<Page Down>) — на страницу вперед.
- **Page Down with Selection** (<Shift>+<Page Down>) — на страницу вперед вместе с выделением.
- **Page Up** (<Page Up>) — на страницу вверх.
- **Page Up with Selection** (<Shift>+<Page Up>) — на страницу вверх вместе с выделением.
- **Move Caret to Previous Word** (<Ctrl>+<Left>) — курсор к предыдущему слову.
- **Move Caret to Previous Word with Selection** (<Ctrl>+<Shift>+<Left>) — курсор к предыдущему слову вместе с выделением.
- **Right** (<Right>) — курсор вправо.
- **Right with Selection** (<Shift>+<Right>) — курсор вправо вместе с выделением.
- **Scroll Down** (<Ctrl>+<Down>, <Ctrl>+<Shift>+<Down>) — окно редактора вниз.
- **Scroll to Center** (<Ctrl>+<M>) — отцентровать окно редактора.
- **Scroll Up** (<Ctrl>+<Up>, <Ctrl>+<Shift>+<Up>) — окно редактора вверх.
- **Select Line at Caret** — выделить текущую строку.
- **Select Word at Caret** (<Ctrl>+<W>) — выделить слово под курсором. Данная команда имеет дополнительные возможности, а не просто выделяет слово под курсором. Если выполнить эту команду повторно, область выделения будет расширена в соответствии с границами кодовой конструкции (рис. 3.125).

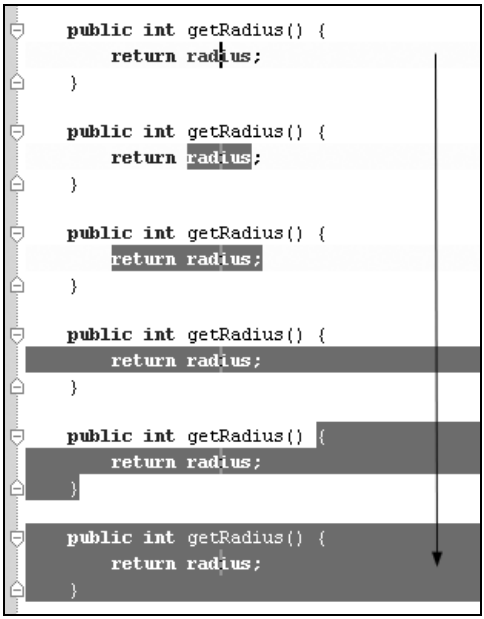
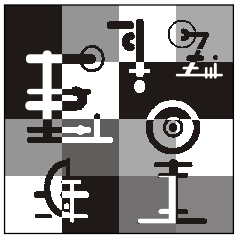


Рис. 3.125. Работа команды выделения слова под курсором

- **Split Line** (<Ctrl>+<Enter>) — разделить строку. К разделенной строке применяются стили форматирования кода (*см. раздел 4.1.7*).
- **Start new Line** (<Shift>+<Enter>) — начать новую строку.
- **Tab** (<Tab>) — символ табуляции.
- **Move Caret to Text End** (<Ctrl>+<End>) — курсор к концу текста.
- **Move Caret to Text End with Selection** (<Ctrl>+<Shift>+<End>) — курсор к конце текста вместе с выделением.
- **Move Caret to Text Start** (<Ctrl>+<Home>) — курсор к началу текста.
- **Move Caret to Text Start with Selection** (<Ctrl>+<Shift>+<Home>) — курсор к началу текста вместе с выделением.
- **Toggle Case** (<Ctrl>+<Shift>+<U>) — сменить регистр букв.
- **Column Mode** (<Alt>+<Shift>+<Insert>) — переключить режим выделения (*см. раздел 3.2.2.8*).
- **Toggle Insert/Overwrite** (<Insert>) — переключение режима ввода (*см. раздел 3.5*).
- **Show Whitespaces** — показать пробельные символы (*см. раздел 3.2.4.3*).
- **Unselect Word at Caret** (<Ctrl>+<Shift>+<W>) — снять выделение со слова под курсором. Команда, обратная команде **Select Word at Caret** (см. рис. 3.125).

- **Unindent Selection** (<Shift>+<Tab>) — убрать отступ красной строки с выделенного блока.
- **Up** (<Up>) — курсор вверх.
- **Up with Selection** (<Shift>+<Up>) — курсор вверх вместе с выделением.
- **Next Parameter** (<Tab>) — следующий параметр. Команда работает при отображенной информации о параметрах вызова метода (*см. раздел 3.2.4.8*).
- **Next Template Variable** (<Tab>, <Enter>) — следующая переменная "живого шаблона". Команда работает при вызове "живого шаблона" (*см. разделы 4.1.13, 6.3.2*).
- **Prev Parameter** (<Shift>+<Tab>) — предыдущий параметр. Команда работает при отображенной информации о параметрах вызова метода (*см. раздел 3.2.4.8*).
- **Previous Template Variable** (<Shift>+<Tab>) — предыдущая переменная "живого шаблона". Команда работает при вызове "живого шаблона" (*см. разделы 4.1.13, 6.3.2*).

Можно заметить, что некоторые команды используют одинаковые сочетания клавиш вызова. Это не ошибка в программе. Обычно такие команды не могут быть вызваны одновременно.



Глава 4

Настройки

Все настройки IntelliJ IDEA собраны в одном большом диалоге (рис. 4.1). Настройки можно вызывать как с помощью нажатия на соответствующие кнопки с пиктограммами, так и с помощью "горячих" клавиш, указанных рядом с подписями в круглых скобках.

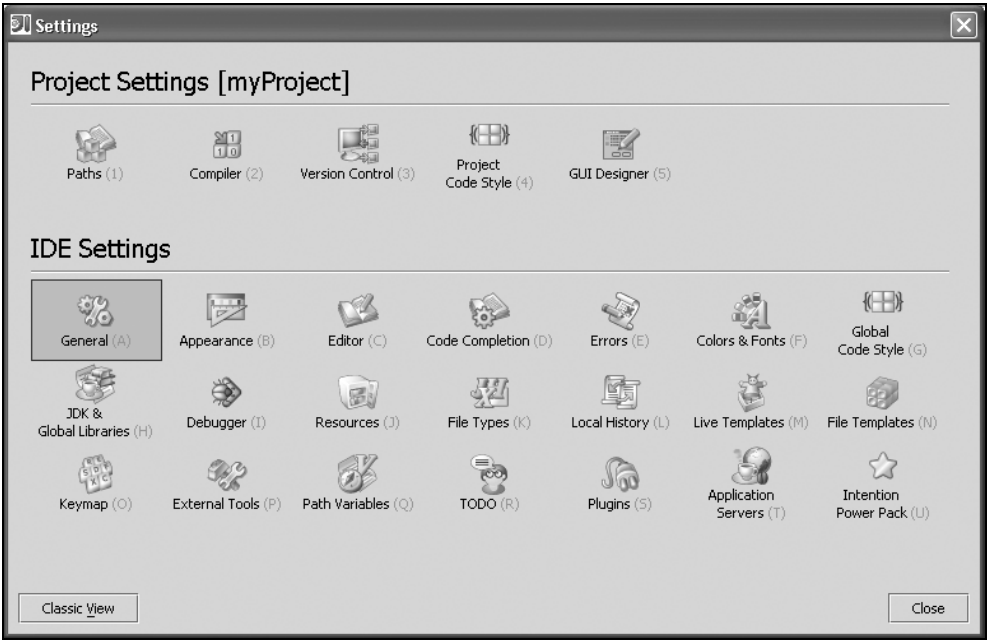


Рис. 4.1. Диалог настроек IntelliJ IDEA

Внутреннее содержимое диалога разделено на две части:

- **Project Settings** (Настройки проекта) — в случае, если открыт какой-то проект. Если ни одного проекта не открыто, эта часть диалога называется **Template Project Settings** (Настройка шаблона проекта);
- **IDE Settings** (Настройки IDE).

Диалог настроек имеет две формы отображения:

- **Default View** (Вид по умолчанию), с размещением кнопок настройки на одной панели (см. рис. 4.1);
- **Classic View** (Классический вид), с двумя закладками, соответствующими общим настройкам и настройкам проекта, внутри которых имеется список опций (рис. 4.2).

Переключение между двумя этими видами осуществляется с помощью кнопок, расположенных в левой нижней части диалога, **Classic View** (Классический вид) и **Default View** (Вид по умолчанию).

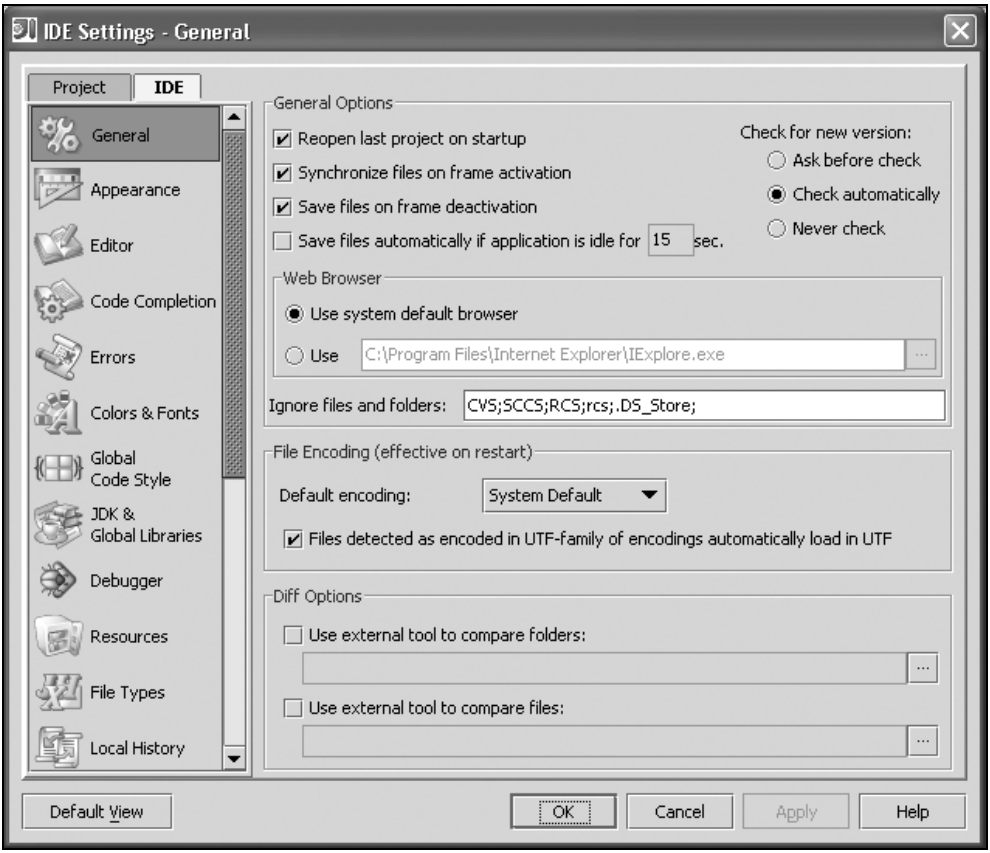


Рис. 4.2. Классический вид диалога настроек

Диалог настроек имел классический вид в прошлых версиях IntelliJ IDEA до версии 4.0 и присутствует в интерфейсе специально для тех программистов, которые привыкли к нему.

Вообще в IntelliJ IDEA есть три уровня настроек:

1. Общие настройки, актуальные как для самой IntelliJ IDEA, так и для всех проектов.
2. Настройки проекта.
3. Настройки модуля.

Настройки проекта могут перекрывать или менять некоторые из глобальных настроек. К таким настройкам относятся, например, настройки стиля кода для проекта.

Для работы с первыми двумя уровнями настроек используется диалог, представленный на рис. 4.1. Настройки модулей спрятаны внутри настроек путей проекта (левая верхняя кнопка в диалоге настроек).

Далее мы подробно рассмотрим все возможные настройки IntelliJ IDEA.

4.1. *IDE Settings (Настройки IDE)*

Существуют следующие настройки IDE:

- **General** (Общие) — настройки, для которых не удалось установить принадлежность к другим типам настроек;
- **Appearance** (Внешний вид) — настройки общих параметров интерфейса пользователя IntelliJ IDEA;
- **Editor** (Редактор) — настройки редактора IntelliJ IDEA;
- **Code Completion** (Завершение кода) — настройки функции завершения кода;
- **Errors** (Ошибки) — настройки отображения информации об ошибках и инспекциях кода;
- **Colors & Fonts** (Цвета и Шрифты) — настройка цветов и шрифтов в редакторе;
- **Global Code Style** (Глобальные стили кода) — настройка общих стилевых принципов форматирования кода;
- **JDK & Global Libraries** (JDK и Глобальные библиотеки) — настройка JDK, доступных для компиляции и запуска приложений, а также библиотек, доступных всем проектам в IntelliJ IDEA;
- **Debugger** (Отладчик) — настройки отладчика;
- **Resources** (Ресурсы) — настройка соответствия внешних ресурсов локально размещенным файлам;

- **File Types** (Типы файлов) — настройка типов файлов, поддерживаемых IntelliJ IDEA;
- **Local History** (Локальная история) — настройка параметров встроенной локальной системы контроля версий IntelliJ IDEA;
- **Live Templates** ("Живые шаблоны") — настройка "живых шаблонов";
- **File Templates** (Шаблоны файлов) — настройка шаблонов файлов;
- **Keymap** ("Горячие" клавиши) — настройка "горячих" клавиш вызова различных команд IntelliJ IDEA;
- **External Tools** (Внешние программы) — настройка команд для вызова (запуска) различных утилит;
- **Path Variables** (Местоположения) — настройка именованных переменных путей для использования в любом проекте;
- **TODO** (Заметки) — настройка параметров отображения списка комментариев "to do" — "что сделать";
- **Plugins** (Плагины) — настройка дополнительных модулей расширения;
- **Application Servers** (Серверы приложений) — настройка серверов приложений;
- **Intention Power Pack** — настройка параметров плагина Intention Power Pack, поставляемого вместе с IntelliJ IDEA.

Иллюстрации диалога настроек мы будем приводить для режима отображения по умолчанию.

4.1.1. Диалог настроек *General (Общие)*

Диалог общих настроек представлен на рис. 4.3.

Опции диалога общих настроек объединены в три группы:

- **General Options** (Общие настройки) — собственно общие настройки:
  - **Reopen last project on startup** — открывать последний проект при старте IntelliJ IDEA. Опция, остающаяся полезной до тех пор, пока число проектов, с которыми вы работаете, равно одному;
  - **Synchronize files on frame activation** — синхронизировать файлы при активации окна IntelliJ IDEA. Опция позволяет включить автоматическую проверку изменений в файле проекта. В случае, если такие изменения будут найдены, IntelliJ IDEA автоматически их перезагрузит. Если файл был изменен локально, IntelliJ IDEA отобразит соответствующий диалог с вопросом о дальнейших действиях. Если изменившийся файл — это файл проекта или модуля, будет предложено перезагрузить проект, дабы "подхватить" все произошедшие в нем изменения;

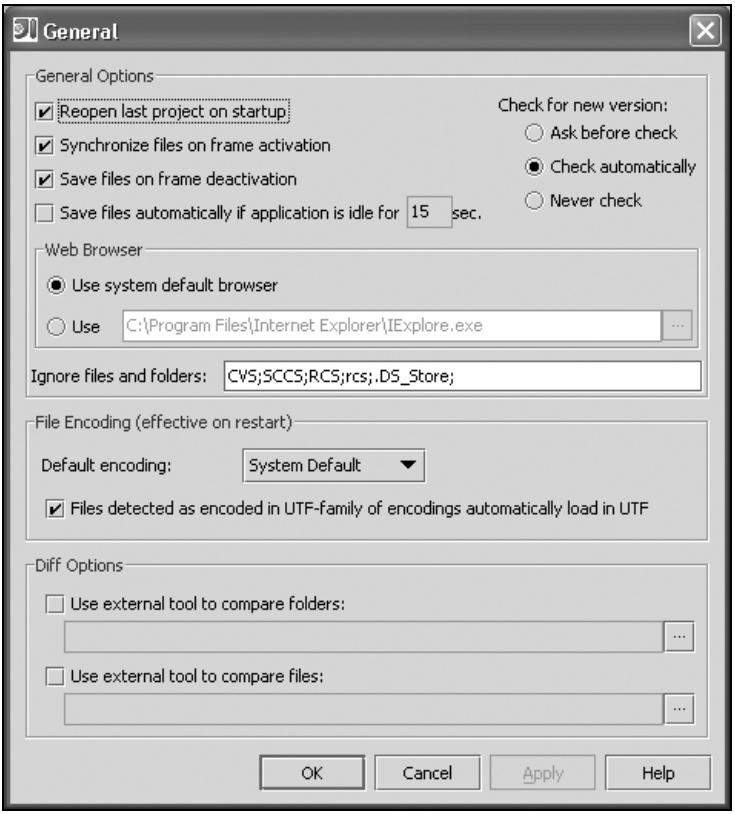


Рис. 4.3. Диалог настроек **General**

- **Save files on frame deactivation** — сохранять файлы, измененные в IntelliJ IDEA при деактивации окна IntelliJ IDEA. Данная опция весьма полезна, т. к. позволяет держать все файлы проекта в целости и сохранности. За все время моей работы с IntelliJ IDEA (а это уже почти два с половиной года) было всего несколько случаев, чтобы содержимое файлов было каким-либо образом испорчено. Да и то это произошло при тестировании версии IntelliJ IDEA, находящейся в стадии разработки;
- **Save files automatically if application is idle for [...] sec.** — сохранять файлы автоматически, если приложение было неактивно столько-то [...] секунд. По умолчанию эта опция выключена, поскольку ее с лихвой компенсирует предыдущая опция — сохранение файлов при деактивации окна IntelliJ IDEA;
- **Check for new version** — проверять наличие новой версии IntelliJ IDEA — весьма полезная функция, поскольку компания JetBrains периодически выпускает обновления версий IntelliJ IDEA с исправлен-

ными ошибками (bug fix). У функции проверки новой версии есть три опции:

- ◊ **Ask before check** — проверять, но спрашивать перед проверкой;
- ◊ **Check automatically** — проверять автоматически;
- ◊ **Never check** — никогда не проверять;

- **Web Browser** — варианты использования Web-браузера. Есть несколько случаев, когда IntelliJ IDEA требуется открывать документы в браузере: просмотр внешней документации JavaDoc, переход по внешней ссылке из встроенной системы помощи, переход по ссылке на информацию о плагине. Есть два варианта вызова Web-браузера:
  - ◊ вызов браузера по умолчанию (**Use system default browser**);
  - ◊ вызов конкретного браузера;
- **Ignore files and folders** — указание IntelliJ IDEA не обращать внимание на перечисленные файлы и каталоги. Как правило, это общие названия, используемые системами контроля версий, а также некоторыми средствами автоматизации;

❐ **File Encoding** (Файловая кодировка) — настройка кодировки файлов. По умолчанию используется системная кодировка, однако имеется возможность ее сменить. Также имеется дополнительная опция распознавать файлы в формате UTF и загружать их именно в этом формате.

❐ **Diff Options** (Свойства утилиты сравнения) — настройка внешних программ для выполнения сравнения. В IntelliJ IDEA имеется встроенная функция сравнения файлов (*см. раздел 12.1*), весьма мощная и удобная. Однако с помощью данных опций можно указать IntelliJ IDEA использовать любые другие программы для сравнения файлов и папок.

4.1.2. Диалог настроек *Appearance (Внешний вид)*

Диалог настроек внешнего вида представлен на рис. 4.4.

В данном диалоге имеется три группы опций:

❐ **UI Options** — настройки интерфейса пользователя:

- **Editor tab placement** — место расположения закладок редактора (см. рис. 3.4). Есть четыре доступных варианта:
  - ◊ **Top** — сверху (выбран по умолчанию);
  - ◊ **Left** — слева;
  - ◊ **Bottom** — снизу;
  - ◊ **Right** — справа;

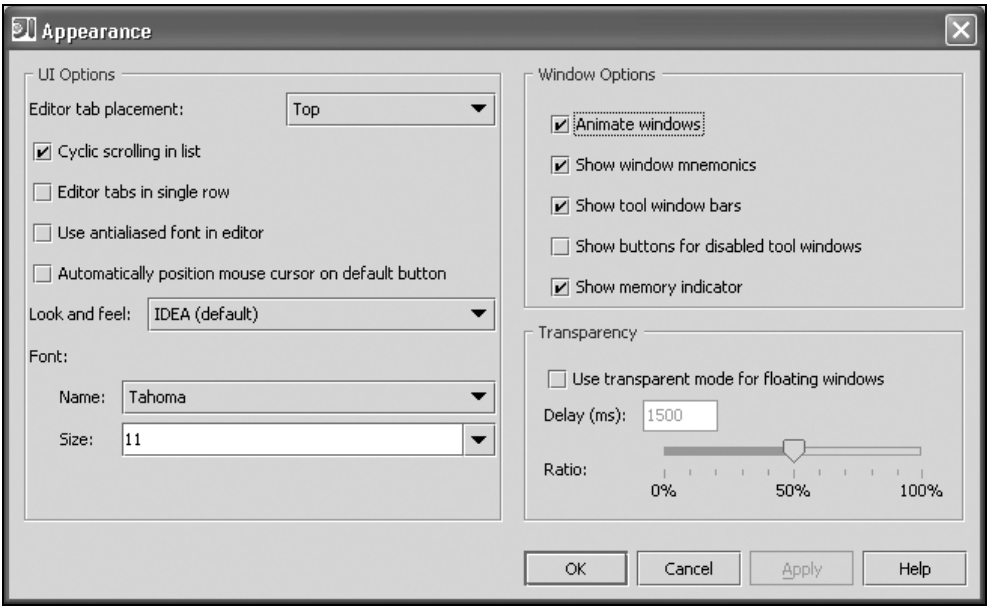


Рис. 4.4. Диалог настроек **Appearance**

- **Cyclic scrolling in list** — возможность циклической прокрутки в списках выбора;
- **Editor tabs in single row** — отображать закладки редактора в одной строке;
- **Use antialiased font in editor** — опция, позволяющая включить сглаживание экранных шрифтов в редакторе. Данная опция полезна не всегда, не везде и не со всяким шрифтом. Например, на мой взгляд, под Windows со стандартными шрифтовыми настройками шрифт без сглаживания выглядит лучше, чем сглаживаемый. Хотя под Mac OS X работать без сглаживания шрифта практически невозможно;
- **Automatically position mouse cursor on default button** — автоматически размещать курсор мыши над кнопкой по умолчанию. Данная функция работает только для диалоговых окон;
- **Look and Feel** — настройки режима Look&Feel. По умолчанию включен Look&Feel специально разработанный JetBrains — **IDEA (default)**. Возможно, именно этот очень симпатичный Look&Feel стал одной из причин высокой популярности среди разработчиков. Также имеется и стандартный набор вариантов Look&Feel (**CDE/Motiff**, **Metal**, **Windows** плюс **Mac OS** при работе под Mac OS X);
- **Font** — настройки типа шрифта и его размера;

- **Window Options** — настройки окон:
  - **Animate windows** — эффекты анимации при работе с окнами, например, слайд-эффект инструментальных окон, постепенное изменение цвета активного или пассивного окна при изменении его статуса и т. п.;
  - **Show window mnemonics** — использование мнемонических сочетаний для вызова инструментальных окон — "горячих" клавиш <Alt>+<цифра>;
  - **Show tool window bars** — отображать поля, расположенные по периметру основного окна IntelliJ IDEA. На этих полях размещаются кнопки быстрого открытия инструментальных окон (см. рис. 3.2);
  - **Show buttons for disabled tool windows** — показывать кнопки быстрого открытия инструментальных окон даже для тех окон, которые в данный момент неактивны;
  - **Show memory indicator** — показывать индикатор использованной памяти, расположенный в правом нижнем углу основного окна IntelliJ IDEA (*см. раздел 3.5*);

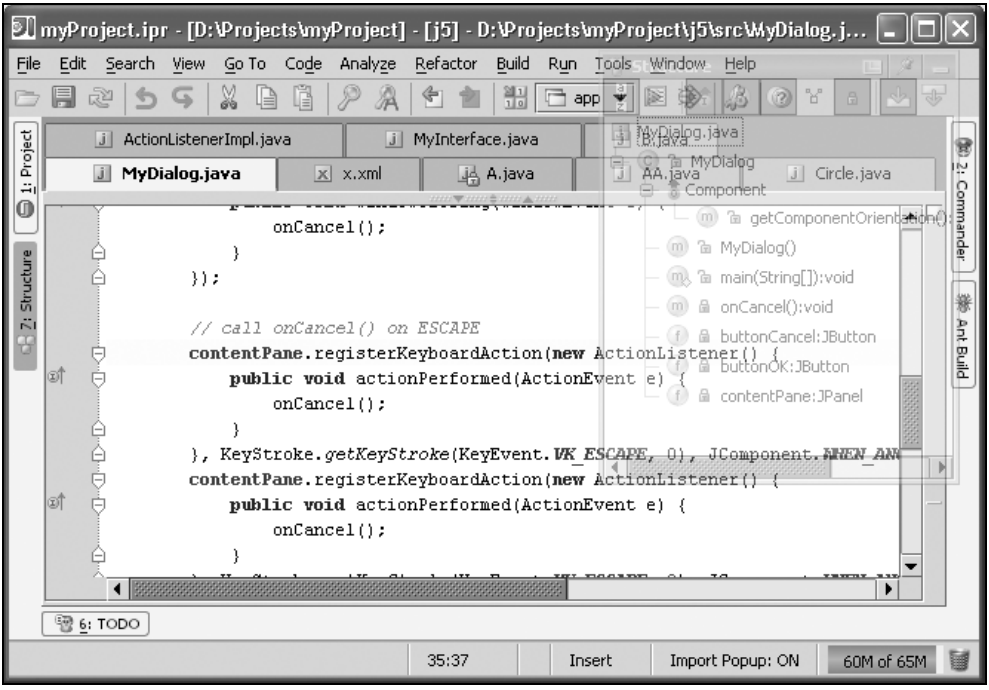


Рис. 4.5. Полупрозрачное инструментальное окно

- **Transparency** — настройка прозрачности. Данная группа содержит опцию включения прозрачности для инструментальных окон, находящихся в пла-

вающем режиме (**Use transparent mode for floating windows**). Если эта опция включена, можно настроить время задержки, спустя которое инструментальные окна будут становиться прозрачными, а также степень их прозрачности (рис. 4.5). Этот режим позволяет держать на экране несколько плавающих окон так, чтобы они не мешали информации в основных окнах редактора.

4.1.3. Диалог настроек *Editor (Редактор)*

Диалог настроек редактора представлен на рис. 4.6.

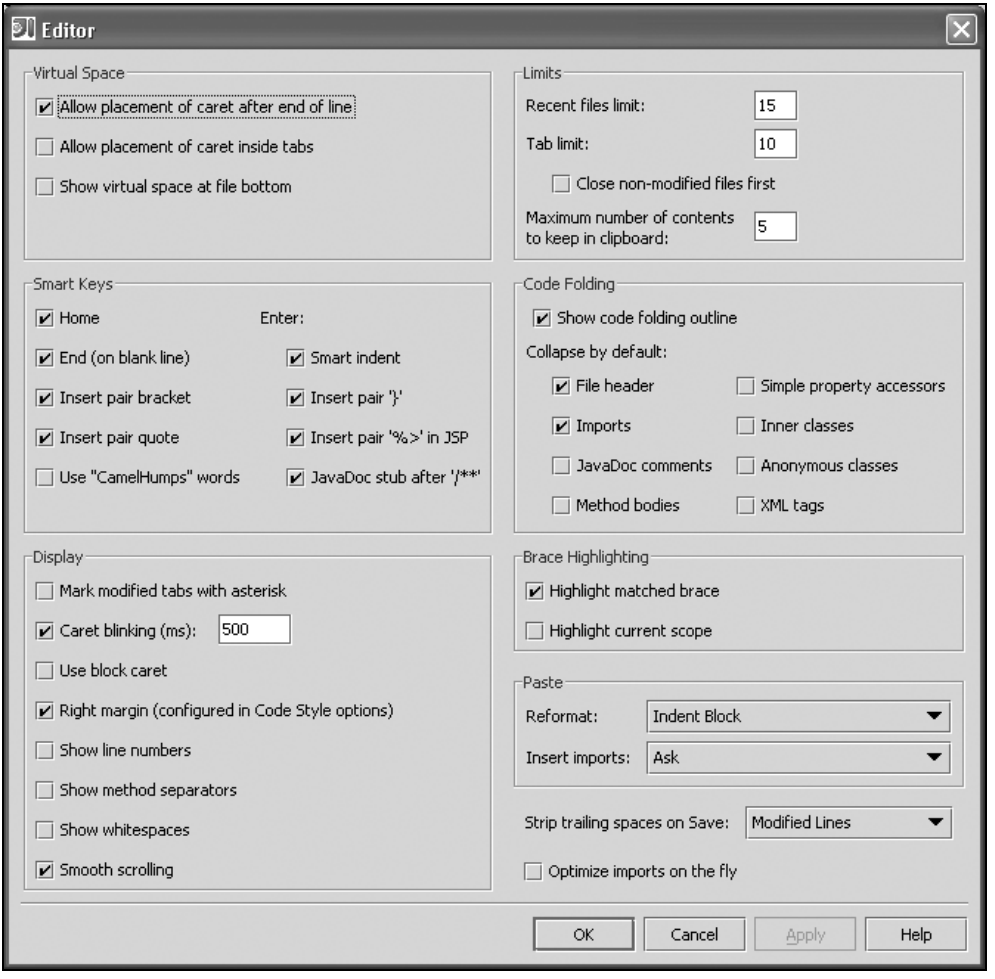


Рис. 4.6. Диалог настроек **Editor**

IntelliJ IDEA имеет очень удобный редактор со множеством настроек на любой вкус. Все опции разделены на восемь групп (вернее на семь плюс пара опций вне объединяющих панелей).

- **Virtual Space** — виртуальное пространство (редактора):
  - **Allow placement of caret after end of line** — разрешить курсору располагаться за концом строки — эта опция позволяет курсору перемещаться по пустому пространству, не заполненному пробелами (рис. 4.7);

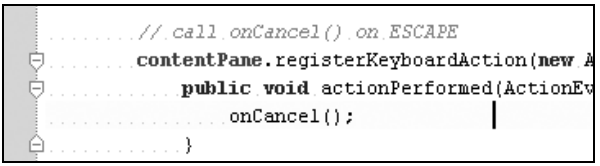


Рис. 4.7. Расположение курсора за концом строки

  - **Allow placement of caret inside tabs** — разрешить курсору располагаться внутри знаков табуляции;
  - **Show virtual space at file bottom** — показывать пустое пространство в конце файла в редакторе.
- **Smart Keys** — "умные" клавиши — включают нестандартное поведение стандартных клавиш. По умолчанию большинство из этих опций включено:
  - **Home** — если опция включена, нажатие клавиши <Home> будет перемещать курсор не в самое начало строки, а в позицию слева, где заканчиваются пробельные символы. Повторное нажатие клавиши <Home> переместит курсор в самое начало строки;
  - **End (on blank line)** — если эта опция включена, нажатие клавиши <End> на пустой строке переведет курсор в позицию красной строки, установленной в соответствии с текущими настройками стиля кода (*см. раздел 4.1.7*);
  - **Insert pair bracket** — когда эта опция включена, IntelliJ IDEA будет автоматически вставлять закрывающую скобку, если в редакторе печатается открывающая. Опция действует на открывающую скобку: круглую и квадратную;
  - **Insert pair quote** — автоматически вставлять парную кавычку. Команда действует на одинарную и двойную кавычку;
  - **Use "CamelHumps" words** — включение этой опции приведет к тому, что IntelliJ IDEA будет распознавать части названий переменных, методов и классов, содержащих заглавные буквы и знаки подчеркивания, например, `ActionListenerImpl` или `m_projetProperty`. Данная

опция действует при командах перемещения курсора внутри окна редактора:

- ◊ <Ctrl>+<Right>, <Ctrl>+<Left> — перемещение вправо-влево между структурными элементами, также совместно с нажатой клавишей <Shift> для выделения;
- ◊ <Ctrl>+<W> — структурное выделение слов и выражений;
- ◊ <Ctrl>+<Backspace>, <Ctrl>+<Delete> — удаление слова, начиная с начала и с конца;
- ◊ а также двойной щелчок мыши на названиях;

- **Enter** — подгруппа опций, управляющих вводом после нажатия клавиши <Enter>:
  - ◊ **Smart indent** — "умная" установка красной строки, в соответствии с настройками стиля кода;
  - ◊ **Insert pair '}'** — вставка парной закрывающей фигурной скобки;
  - ◊ **Insert pair '%>' in JSP** — вставка парного закрывающего тега JSP;
  - ◊ **JavaDoc stub after '/**'** — добавление шаблона комментария JavaDoc после набора комментария, начинающегося с '/**'. Эта возможность вообще чрезвычайно удобна, поскольку большинство комментариев, начинающихся с '/**', пишутся в формате JavaDoc. А когда IntelliJ IDEA автоматически вставляет шаблон данного комментария, остается только его заполнить. Особенно это удобно для функций с большим числом параметров или выбрасываемых исключений.

Опции данной группы включены по умолчанию. Не помню случая, чтобы мне пришло в голову выключить какую-нибудь из них.

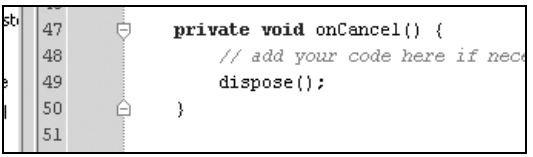


Рис. 4.8. Номера строк на полях редактора

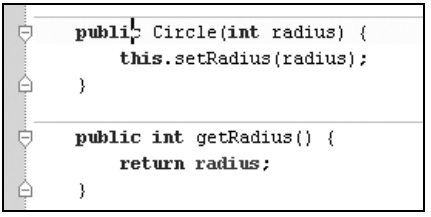


Рис. 4.9. Разделители методов

- **Display** (Отображение):
  - **Mark modified tabs with asterisk** — помечать закладки с редакторами, в которых произошли какие-либо изменения, с помощью звездочки '*'. Данная опция выключена по умолчанию, поскольку IntelliJ IDEA берет на себя сохранение измененных файлов (если вы не выключили эту возможность);
  - **Caret blinking (ms)** — частота мигания курсора (в миллисекундах);
  - **Use block caret** — использовать значок курсора в виде блока. Изначально эта опция выключена, при этом курсор в редакторе выглядит как тонкая вертикальная черта, и превращается в блок, только если включен режим замены (смена режимов управляется клавишей <Insert>);
  - **Right margin** — отображать тонкую вертикальную линию, для обозначения правой границы в редакторе (количество символов в строке задается в настройках стиля кода);
  - **Show line numbers** — показывать номера строк (рис. 4.8);
  - **Show method separators** — показывать разделители между методами (не включая абстрактные методы) в виде тонкой черты (рис. 4.9);
  - **Show whitespaces** — отображать пробелы в виде небольших точек (*см. раздел 3.2.4.3*, рис. 3.32);
  - **Smooth scrolling** — "мягкая" прокрутка.
- **Limits** — ограничения:
  - **Recent files limit** — максимальное количество файлов, открытых в редакторе, запоминаемых в качестве последних открытых файлов (*см. раздел 3.2.4.14*);
  - **Tab limit** — максимальное количество закладок с редакторами, отображаемых одновременно. Данное ограничение может показаться неправильным — а если программисту нужно одновременно работать с большим количеством файлов? Однако, учитывая легкость навигации между различными файлами в IntelliJ IDEA, данная проблема оказывается совершенно несущественной, просто интерфейс не засоряется лишними окнами. У данного ограничения есть дополнительная опция закрывать неизмененные файлы первыми (**Close non-modified files first**);
  - **Maximum number of contents to keep in clipboard** — максимальное количество частей текста, помещенных в буфер обмена (*см. раздел 3.2.2.7*).
- **Code Folding** — свернутое/развернутое представление кода:
  - **Show code folding outline** — показывать специальные визуальные компоненты для управления свернутое/развернутым представлением (*см. раздел 3.2.4.19*, рис. 3.45);

- **Collapse by default** — сворачивать по умолчанию:
  - ◊ **File header** — заголовок файла (обычно, это некоторый вступительный комментарий или строка авторских прав);
  - ◊ **Imports** — строки импортов;
  - ◊ **JavaDoc comments** — комментарии JavaDoc;
  - ◊ **Method bodies** — тела методов;
  - ◊ **Simple property accessors** — простые методы доступа к свойствам класса — методы `get` и `set`;
  - ◊ **Inner classes** — внутренние классы;
  - ◊ **Anonymous classes** — анонимные классы;
  - ◊ **XML tags** — теги XML.

❐ **Brace Highlighting** — подсветка парных скобок:

- **Highlight matched brace** — подсвечивать парные скобки (рис. 4.10);

```
final Circle circle = (Circle) o;

if (getRadius() != circle.getRadius()) return false;
```

Рис. 4.10. Подсветка парных скобок

- **Highlight current scope** — подсвечивать текущий контекст. IntelliJ IDEA по умолчанию подсвечивает текущий контекст (рис. 4.11) с помощью скобки на полях редактора, если курсор наведен на одну из ограничивающих его фигурных скобок. Данная опция позволяет все время видеть эту фигурную скобку, если курсор находится внутри данного контекста.

```
public Circle(int radius) {
    this.setRadius(radius);
}
```

Рис. 4.11. Подсветка текущего контекста

❐ **Paste** — вставка:

- **Reformat** — переформатировать. IntelliJ IDEA позволяет автоматически форматировать блок текста, вставляемый из буфера обмена в соответствии с тем контекстом, куда он попадает. Форматирование производится в соответствии с настройками стиля кода. Всего есть четыре варианта такого форматирования:
  - ◊ **None** — не форматировать;

- ◊ **Indent Block** — поменять отступ красной строки для всего блока;
- ◊ **Indent each line** — поменять отступ красной строки для каждой строки из блока;
- ◊ **Reformat block** — полностью переформатировать блок в соответствии с контекстом и настройками стиля кода;

- **Insert imports** — вставка импортов для классов, используемых во вставляемом блоке в случае, если в новом контексте они не импортированы. Есть три варианта такого импорта:
  - ◊ **All** — автоматическая вставка импортов;
  - ◊ **Ask** — перед вставкой каждой строки импорта будет запрошено подтверждение;
  - ◊ **None** — не вставлять автоматически строки импорта.

❐ И последняя группа из двух опций:

- **Strip trailing spaces on Save** — удалять пробелы в конце строк при сохранении. Данная опция имеет три возможных значения:
  - ◊ **Modified lines** — удалять лишние пробелы в конце изменившихся строк;
  - ◊ **All** — удалять все лишние пробелы в конце всех строк;
  - ◊ **None** — не удалять лишние пробелы в конце строк.
- **Optimize imports on the fly** — оптимизировать строки импорта "на лету". Если данная опция выключена, оптимизировать строки импорта можно в ручном режиме с помощью соответствующей команды (*см. раздел 3.2.11.2*).

4.1.4. Диалог настроек *Code Completion (Завершение кода)*

Диалог настроек опций завершения кода представлен на рис. 4.12.

В диалоге выделено три группы настроек. Больше всего опций в группе настроек завершения кода (**Code Completion**):

❐ **Code Completion** — завершение кода:

- **Case sensitive completion** — завершение кода с учетом регистра. У данной опции есть три возможных значения:
  - ◊ **All** — учитывать регистр всех символов завершаемой строки. Например, если есть поле класса `myLocalPane`, а в редакторе написано `mylocal` — IntelliJ IDEA не найдет ни одного варианта завершения, поскольку '`local`' в имени нужно было писать с большой буквы;
  - ◊ **No** — регистр символов не учитывается;

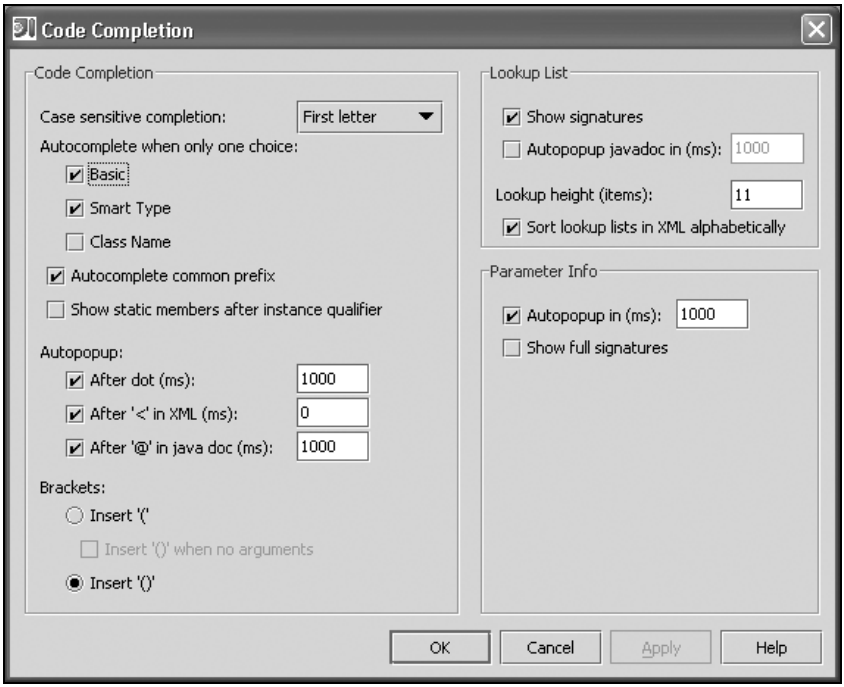


Рис. 4.12. Диалог настроек **Code Completion**

◊ **First letter** — учитывается регистр только первой буквы. Данная опция позволяет IntelliJ IDEA более точно выдавать список возможных вариантов завершения кода в случае, если часть кода для завершения начинается с большой или маленькой буквы. Например, у вас есть некоторый класс, название которого начинается на букву 'A', а в этом классе есть поле, начинающееся на 'a' (листинг 4.1). Если в теле метода `foo()` написать строчную букву 'a' и выполнить команду **Code | Complete | Basic** (Код | Завершить | Простое завершение), IntelliJ IDEA предложит набор вариантов для завершения, в котором сам класс `A` не будет представлен (рис. 4.13). Если же написать заглавную букву 'A', класс `A` будет присутствовать в списке;

Листинг 4.1

```
public class A {
    private int aarbuz;

    public void foo () {

    }
}
```

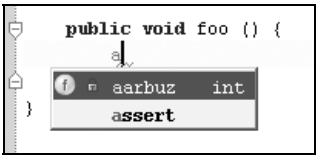


Рис. 4.13. Простое завершение кода — имени класса нет в списке вариантов

- **Autocomplete when only one choice** — завершать автоматически в случае одного варианта завершения. Данная подгруппа имеет три опции для трех различных типов завершения кода: **Basic** (Простое завершение), **Smart Type** ("Умное" завершение типа), **Class Name** (Завершение имени класса). По умолчанию опция для завершения имени класса выключена, и неспроста. Дело в том, что варианты завершения имени класса, которые может найти IntelliJ IDEA, совершенно не подходят программисту (например, в случае, если в проекте отсутствует какая-либо библиотека, и среди вариантов завершения просто нет имени класса из этой библиотеки). Вероятность ошибки слишком высока, а цена ее может быть слишком большой, чтобы принять такую ответственность;
- **Autocomplete common prefix** — завершать автоматически общий префикс (рис. 4.14). Если все варианты завершения имеют общий префикс (на рис. 4.14 — `Component`), этот префикс будет автоматически подставлен в текст;

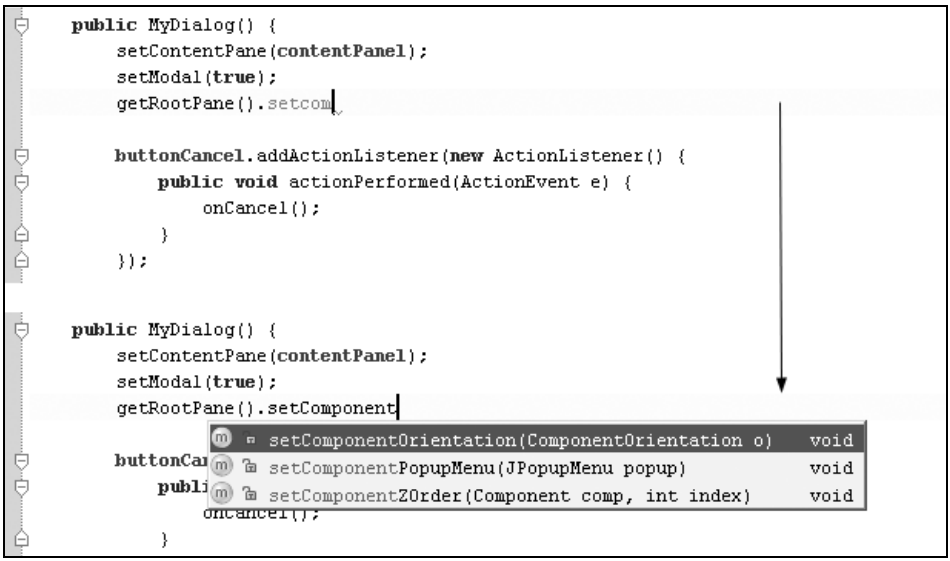


Рис. 4.14. Автоматическое завершение общего префикса

- **Show static members after instance qualifier** — если эта опция установлена, при вызове команды завершения для экземпляра какого-либо класса IntelliJ IDEA будет включать в список вариантов завершения и статические методы данного класса;
- **Autopopup** — автоматические всплывающие подсказки о завершении кода. В данной группе находятся три опции, для которых можно задать время задержки в миллисекундах:
  - ◊ **After dot (ms)** — после точки IntelliJ IDEA будет автоматически предлагать список из полей и методов класса;
  - ◊ **After '<' in XML (ms)** — после знака '<' в XML-файле;
  - ◊ **After '@' in java doc (ms)** — после знака '@' в комментариях JavaDoc;
- **Brackets** — скобки. Опция имеет два возможных значения:
  - ◊ **Insert '('** — вставлять одиночную скобку '('. При этом значении IntelliJ IDEA не будет вставлять закрывающую скобку для автоматически завершенных названий методов, а вставит только открывающую. При этом у данного значения есть дополняющая опция: **Insert '()' when no arguments** — вставлять обе скобки, когда у метода нет аргументов;
  - ◊ **Insert '()'** — вставлять двойные скобки. При этом значении IntelliJ IDEA всегда будет вставлять обе скобки при автоматическом завершении названий методов.

Замечание

Независимо от настроек данной опции, IntelliJ IDEA будет всегда вставлять обе скобки после названия метода, если этот метод не возвращает значения.

❐ **Lookup List** — группа опций, задающих свойства списка возможных вариантов завершения:

- **Show signatures** — показывать сигнатуру методов, полей класса и переменных в списке. Когда сигнатуры отображаются, среди них намного проще выбрать правильный вариант;
- **Autopopup javadoc in (ms)** — возможность автоматически отображать комментарий JavaDoc для метода, класса или поля класса, на котором в данный момент находится подсветка. Если данная опция не выбрана, JavaDoc, тем не менее, посмотреть можно. Для этого можно пользоваться сочетанием "горячих" клавиш для вызова команды **Quick JavaDoc** (Быстрый JavaDoc) из меню **View** (Просмотр) — <Ctrl>+<Q>;
- **Lookup height (items)** — количество пунктов в списке на завершение кода;
- **Sort lookup lists in XML alphabetically** — сортировать в алфавитном порядке варианты в списке на завершение в XML.

- **Parameter Info** — группа из двух опций, задающих настройки показа информации о параметрах (рис. 4.15):
  - **Autopopup in (ms)** — автоматическое появление информации о параметрах через заданное количество миллисекунд;
  - **Show full signatures** — показывать полную сигнатуру параметров метода.

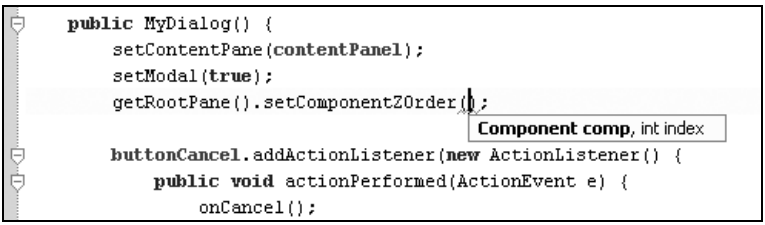


Рис. 4.15. Автоматическое всплывающее окно с информацией о параметрах

4.1.5. Диалог настроек *Errors (Ошибки)*

Настройка параметров ошибок производится с помощью диалога, представленного на рис. 4.16.

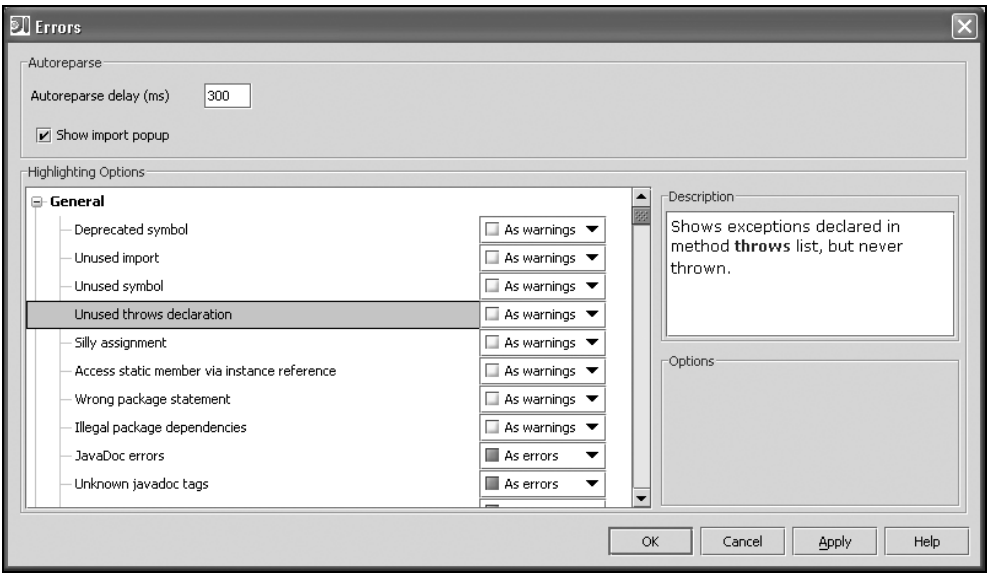


Рис. 4.16. Настройка параметров отображения ошибок и недочетов

Опции разделены на две группы:

- **Autoreparse** (Автоматический анализ) — настройки автоматического анализа кода с целью обнаружения ошибок. Данная группа содержит всего две опции:
  - **Autoreparse delay** (Задержка автоматического анализа) — время в миллисекундах, спустя которое начинается автоматический анализ кода;
  - **Show import popup** (Показывать всплывающие подсказки об импорте) — опция, включающая автоматические подсказки с предложением об импорте классов, используемых в коде, но не присутствующих в списке импорта.
- **Highlighting Options** (Опции подсветки) — настройки подсветки ошибок в коде. Данная группа содержит список всех возможных инспекций, сгруппированных в логические группы, а также панель с их описанием (**Description**) и опции каждого пункта проверки (**Options**), если они у него есть.

Для каждой инспекции может быть настроено, в каком качестве она будет засчитана IntelliJ IDEA при положительных результатах:

- **As errors** — как ошибка (код будет подсвечен красным);
- **As warnings** — как замечание (код будет подсвечен желтым);
- **Do not show** — не помечать.

Первые две группы обеспечиваются собственной функциональностью IntelliJ IDEA, остальные группы инспекций поддерживаются плагином Inspection Gadgets, поставляемым вместе с IntelliJ IDEA. Подробнее о плагинах можно прочитать в *главе 15*.

Далее мы приведем список групп инспекций. Их подробное описание содержится в *разделе 6.4*. Всего имеется 26 групп инспекций:

- **General** (Общие) — группа общих инспекций (*см. раздел 6.4.4.2*);
- **Local Code Analysis** (Локальный анализ кода) — группа инспекций, проверяющих локальный код (*см. раздел 6.4.4.3*);
- **Abstraction issues** (Проблемы абстракции) — группа инспекций, выявляющих проблемы с абстракциями (*см. раздел 6.4.4.4*);
- **Class metrics** (Метрики класса) — группа инспекций, проверяющих метрики класса (*см. раздел 6.4.4.5*);
- **Class structure** (Структура класса) — инспекции, проверяющие структуру класса (*см. раздел 6.4.4.6*);
- **Cloning issues** (Проблемы клонирования) — группа инспекций, занимающихся проблемами клонирования объектов (*см. раздел 6.4.4.7*);
- **Code maturity issues** (Проблемы устаревшего кода) — инспекции, выявляющие код, который, возможно, является устаревшим (*см. раздел 6.4.4.8*);

- **Code style issues** (Стилевые проблемы) — инспекции, проверяющие стиль кода (*см. раздел 6.4.4.9*);
- **Encapsulation issues** (Проблемы инкапсуляции) — инспекции, занимающиеся проверкой нарушений инкапсуляции (*см. раздел 6.4.4.10*);
- **Error handling** (Проблемы обработки ошибок) — инспекции, проверяющие возможные проблемы при обработке и диагностике ошибок (*см. раздел 6.4.4.11*);
- **Finalization issues** (Проблемы финализации) — группа инспекций, проверяющих код на возможные проблемы в блоках финализации (`try {} finally {}`) (*см. раздел 6.4.4.12*);
- **Imports** (Проблемы импорта) — инспекции, проверяющие строки импорта (*см. раздел 6.4.4.13*);
- **Initialization issues** (Проблемы инициализации) — инспекции, исследующие проблемы инициализации (*см. раздел 6.4.4.14*);
- **Internationalization issues** (Проблемы интернационализации) — инспекции, исследующие проблемы интернационализации кода (*см. раздел 6.4.4.15*);
- **JUnit issues** (Проблемы тестов JUnit) — инспекции, проверяющие ошибки в тестах JUnit (*см. раздел 6.4.4.16*);
- **Method metrics** (Метрики методов) — инспекции, проверяющие метрики методов (*см. раздел 6.4.4.17*);
- **Naming conventions** (Соглашения об именах) — инспекции, проверяющие соблюдения соглашений об именах (*см. раздел 6.4.4.18*);
- **Performance issues** (Проблемы производительности) — инспекции, проверяющие наличие возможных проблем с производительностью (*см. раздел 6.4.4.19*);
- **Portability issues** (Проблемы переносимости) — инспекции, проверяющие наличие возможных проблем с переносимостью кода на другие операционные системы (*см. раздел 6.4.4.20*);
- **Potentially confusing code constructs** (Потенциальные проблемные конструкции кода) — инспекции, проверяющие конструкции кода на возможные проблемы (*см. раздел 6.4.4.21*);
- **Probable bugs** (Возможные ошибки) — инспекции, обнаруживающие возможные ошибки в коде (*см. раздел 6.4.4.22*);
- **Security issues** (Проблемы защиты и безопасности) — инспекции, занимающиеся проверкой возможных проблем с защитой и безопасностью (*см. раздел 6.4.4.23*);
- **Serialization issues** (Проблемы сериализации) — инспекции, проверяющие наличие проблем с сериализацией объектов (*см. раздел 6.4.4.24*);

- **Threading issues** (Проблемы многопоточности) — инспекции, исследующие проблемы с многопоточностью (*см. раздел 6.4.4.25*);
- **Verbose or redundant code constructs** (Громоздкие или излишние конструкции кода) — инспекции, проверяющие код на наличие избыточных, слишком сложных и громоздких конструкций (*см. раздел 6.4.4.26*);
- **Visibility issues** (Проблемы видимости) — инспекции, проверяющие проблемы с областью видимости (*см. раздел 6.4.4.27*).

4.1.6. Диалог настроек *Colors & Fonts (Цвета и Шрифты)*

Диалог настроек цветов и шрифтов представлен на рис. 4.17.

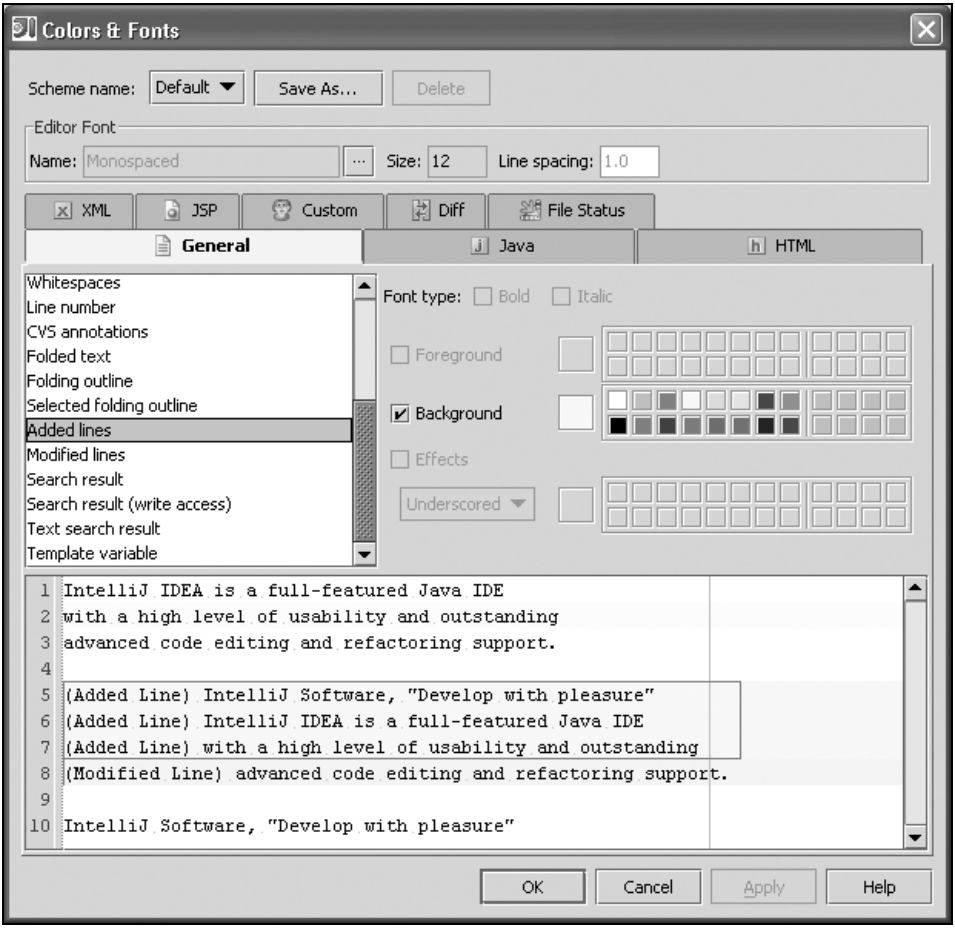


Рис. 4.17. Диалог настроек **Colors & Fonts**

С помощью этого диалога можно настроить цвета всех компонентов текста в редакторе. В данном диалоге также можно установить параметры шрифта на панели **Editor Font** (Шрифт редактора):

- тип шрифта (**Name**);
- размер шрифта (**Size**);
- а также межстрочное расстояние (**Line spacing**).

Параметры цветов и шрифта могут быть сохранены в *цветовую схему*, с помощью кнопки **Save as** (Сохранить как) в верхней части диалога. По умолчанию используется схема **Default** (По умолчанию). Чтобы изменить какие-либо настройки, нужно сначала сохранить текущую схему с новым именем, поскольку изменять схему **Default** (По умолчанию) нельзя (рис. 4.18).

Рис. 4.18. Сообщение о необходимости сохранить схему с другим именем для изменения

Цветовые настройки разделены на восемь частей, каждой из которых соответствует своя закладка в диалоге:

1. **General** — общие настройки.
2. **Java** — цветовые параметры при редактировании Java-файлов.
3. **HTML** — цветовые параметры при редактировании HTML-файлов.
4. **XML** — цветовые параметры при редактировании XML-файлов.
5. **JSP** — цветовые параметры при редактировании HTML-файлов.
6. **Custom** — пример создания цветовых настроек для пользовательского типа файлов.
7. **Diff** — цветовые параметры на панели сравнения.
8. **File Status** — цвета файлов, соответствующие разным статусам файла в системе контроля версий.

Далее мы подробно рассмотрим составляющие из данных восьми закладок.

В левой части каждой закладки содержится список различных компонент, для которых можно настроить собственные цветовые параметры. Если выбрать из этого списка ту или иную компоненту, то на панели в нижней части закладки она будет выделена рамкой (см. рис. 4.17).

В нижней части закладок можно увидеть, как будет выглядеть раскрашиваемое содержимое в тех или иных цветах. Каждый элемент из этого поля также является и ссылкой на компоненты, для которых можно задать цвет.

В правой верхней части закладки содержатся:

- две опции настройки шрифта (**Font type**):
  - **Bold** — жирный шрифт;
  - **Italic** — наклонный шрифт (курсив);
- две опции настройки цвета:
  - **Foreground** — цвет переднего плана. Для текстовых элементов — это цвет текста;
  - **Background** — цвет фона;
- опция дополнительного эффекта (**Effect**) текста, а также его (эффекта) цвет:
  - **Underscored** — подчеркнутый;
  - **Underwaved** — подчеркнутый волнистой линией;
  - **Bordered** — обрамленный;
  - **Strikeout** — перечеркнутый;
- цветовые настройки, а также дополнительные параметры шрифта, которые можно задать для некоторых компонент.

Некоторые компоненты имеют ограничения по настройке, например, общая настройка цвета фона (**General | Background**) имеет возможность настройки только цвета фона и т. п. Для таких компонент мы будем отдельно это указывать в скобках. Если по поводу ограничений для настройки компоненты ничего не указано, значит, для нее доступны все возможные настройки.

4.1.6.1. Закладка *General (Общие)*

Закладка имеет следующие опции:

- **Background** — цвет фона в редакторе (только цвет фона);
- **Background in readonly files** — цвет фона в файлах, доступных только для чтения (только цвет фона);
- **Readonly fragment background** — цвет фона фрагмента, доступного только для чтения (только цвет фона);
- **Default text** — текст по умолчанию;
- **Selection** — выделение (только цвета текста и фона);
- **Caret** — курсор (только цвет переднего плана);
- **Caret row** — строка, содержащая курсор (только цвет фона);

- **Right margin** — тонкая вертикальная линия, обозначающая правую границу в редакторе (только цвет переднего плана);
- **Whitespaces** — символы пробела (*раздел 3.2.4.3*) (только цвет переднего плана);
- **Line number** — номер строки (только цвет переднего плана);
- **CVS annotations** — аннотация CVS (только цвет переднего плана);
- **Folded text** — свернутый текст (*см. раздел 3.2.4.19*);
- **Folding outline** — маркер свертывания/развертывания текста (*см. раздел 3.2.4.19*) (только цвет переднего плана);
- **Selected folding outline** — выделенный маркер свертывания/развертывания текста (*см. раздел 3.2.4.19*) (только цвет переднего плана);
- **Added lines** — добавленные строки (только цвет фона);
- **Modified lines** — измененные строки (только цвет фона);
- **Search result** — результат поиска (*см. раздел 3.2.3.6*);
- **Search result (write access)** — результат поиска (обращение на запись) (*см. раздел 3.2.3.9*);
- **Text search result** — текстовый результат поиска (*см. раздел 3.2.3.9*);
- **Template variable** — переменная шаблона.

4.1.6.2. Закладка *Java (Java-файлы)*

Закладка имеет следующие опции:

- **Keyword** — ключевое слово;
- **Number** — число;
- **String** — строка;
- **Valid escape in string** — допустимая `escape`-последовательность (последовательность из символа обратной косой черты '\' и другого символа);
- **Invalid escape in string** — недопустимая `escape`-последовательность (последовательность из символа обратной косой черты '\' и другого символа);
- **Operator sign** — знак оператора;
- **Parentheses** — круглые скобки;
- **Braces** — фигурные скобки;
- **Brackets** — квадратные скобки;
- **Comma** — запятая;
- **Semicolon** — точка с запятой;

- **Dot** — точка;
- **Line comment** — строковый комментарий;
- **Block comment** — блочный комментарий;
- **JavaDoc comment** — комментарий JavaDoc;
- **JavaDoc tag** — тег JavaDoc;
- **JavaDoc markup** — HTML-разметка в JavaDoc;
- **Unknown symbol** — неизвестный элемент;
- **Error** — ошибка;
- **Warning** — предупреждение;
- **Deprecated symbol** — устаревший (`@deprecated`) элемент/символ;
- **Unused symbol** — неиспользуемый элемент;
- **Class** — класс;
- **Interface** — интерфейс;
- **Local variable** — локальная переменная;
- **Instance field** — поле класса;
- **Static field** — статическое поле класса;
- **Parameter** — параметр метода;
- **Method call** — вызов метода;
- **Method declaration** — определение метода;
- **Constructor call** — вызов конструктора;
- **Constructor declaration** — определение конструктора;
- **Static method** — статический метод;
- **Todo item** — запись "todo" (заметка) (*см. раздел 3.4.8.7*);
- **Matched brace** — парная скобка;
- **Unmatched brace** — непарная скобка;
- **Bad character** — неправильный, ошибочный символ;
- **Method separator** — тонкая линия, разделяющая методы (*см. раздел 4.1.3*) (только цвет переднего плана);
- **Breakpoint line** — строка с точкой останова;
- **Execution point** — выполняемая строка (при отладке);
- **Annotation name** — имя аннотации;
- **Annotation attribute name** — имя атрибута в аннотации.

4.1.6.3. Закладка *HTML (HTML-файлы)*

Закладка имеет следующие опции:

- **Comment** — комментарий;
- **Tag** — тег (угловые скобки, его определяющие);
- **Tag name** — имя тега;
- **Attribute name** — имя атрибута;
- **Attribute value** — значение атрибута.

4.1.6.4. Закладка *XML (XML-файлы)*

Закладка имеет следующие опции:

- **Prologue** — пролог XML-файла (заголовок, `DOCTYPE`, и т. п.);
- **Comment** — комментарий;
- **Tag** — тег (угловые скобки, его определяющие);
- **Tag name** — имя тега;
- **Attribute name** — имя атрибута;
- **Attribute value** — значение атрибута;
- **Tag data** — содержимое тега.

4.1.6.5. Закладка *JSP (JSP-файлы)*

Закладка имеет следующие опции:

- **Comment** — комментарий;
- **Scripting background** — цвет фона JSP-скрипта (между тегами `<%` и `%>`);
- **Directive background** — цвет фона директивы (между тегами `<%@` и `%>`);
- **Action background** — цвет фона команды (например, `<jsp:useBean…/>`);
- **Directive name** — имя директивы;
- **Action name** — имя команды;
- **Attribute name** — имя атрибута;
- **Attribute value** — значение атрибута.

4.1.6.6. Закладка *Custom (Файлы, заданные пользователем)*

Закладка имеет следующие опции:

- **Keyword1** — ключевое слово первой группы;
- **Keyword2** — ключевое слово второй группы;

- **Keyword3** — ключевое слово третьей группы;
- **Keyword4** — ключевое слово четвертой группы;
- **Number** — число;
- **String** — строка;
- **Line comment** — строчный комментарий;
- **Block comment** — блочный комментарий.

Подробнее о пользовательских файлах рассказано в *разделе 4.1.11*.

4.1.6.7. Закладка *DIFF (Панель сравнения)*

Для панели сравнения возможности по настройке параметров цвета и шрифта несколько ограничены. Здесь можно настроить только цвет переднего плана и цвет фона:

- **Changed** — измененный текст;
- **Conflict** — конфликт между изменениями;
- **Deleted** — удаленный текст;
- **Inserted** — вставленный текст.

Подробнее о панели сравнения рассказано в *разделе 12.1*.

4.1.6.8. Закладка *File Status (Статус файла)*

Закладка имеет следующие опции:

- **Up to date** — файл имеет последнюю версию;
- **Merged with conflicts** — файл был объединен с конфликтами;
- **Deleted from file system** — файл удален из файловой системы;
- **Ignored** — файл игнорируется;
- **Modified** — измененный файл;
- **Merged** — объединенный файл;
- **Added** — добавленный файл;
- **Unknown** — неизвестный файл;
- **Deleted** — удаленный файл (из системы контроля версий).

4.1.7. Диалог настроек *Global Code Style (Глобальные стили кода)*

Диалог глобальных настроек стиля кода представлен на рис. 4.19.

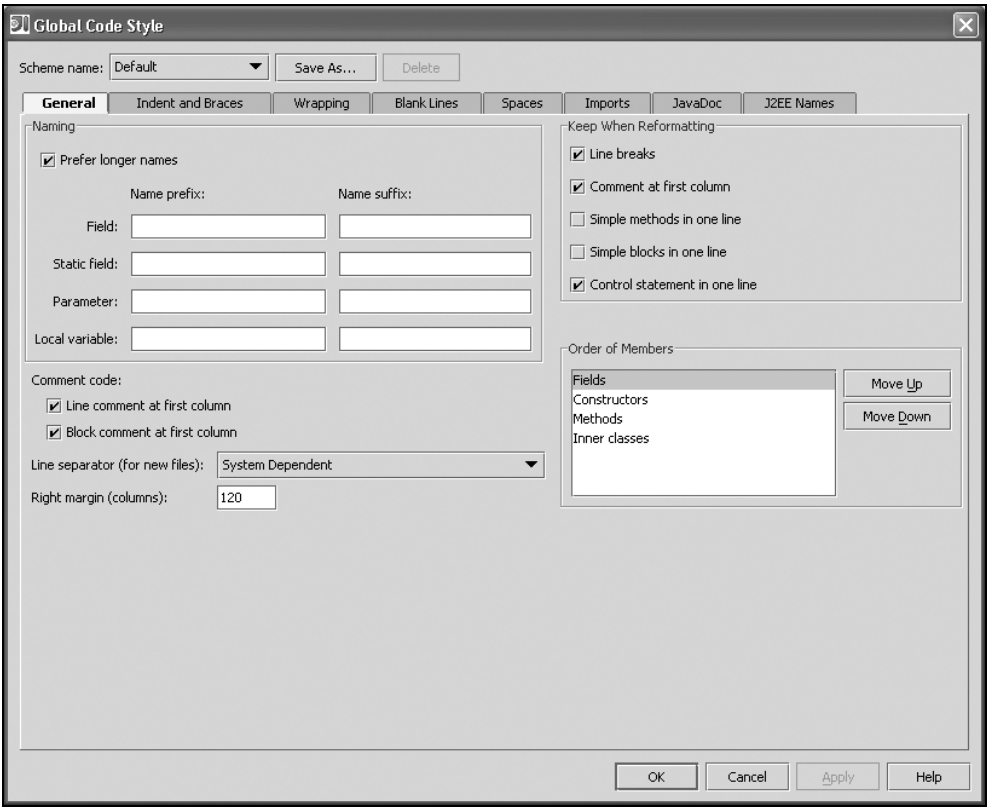


Рис. 4.19. Диалог настроек **Global Code Style**

Все настройки стиля кода разделены на восемь групп:

- **General** — общие;
- **Indent and Braces** — отступ красной строки и фигурные скобки;
- **Wrapping** — перенос;
- **Blank Lines** — пустые строки;
- **Spaces** — пробелы;
- **Imports** — строки импорта;
- **JavaDoc** — комментарии JavaDoc;
- **J2EE Names** — имена J2EE.

Все множество настроек сохранено в виде схемы (**Schema name**). Схемы можно сохранять и удалять с помощью двух кнопок в верхней части диалога **Save As** (Сохранить как) и **Delete** (Удалить). По умолчанию используется схема **Default** (По умолчанию).

Далее мы рассмотрим все группы подробнее.

4.1.7.1. Закладка *General (Общие)*

Общие стилевые настройки разделены на три логические группы плюс несколько дополнительных опций:

- **Naming** (Именование). Данная группа содержит опции именования переменных:
  - **Prefer longer names** — отдавать предпочтения длинным именам. Если данная опция включена, IntelliJ IDEA будет подсвечивать в списке предлагаемых имен более длинное (рис. 4.20, *а*). В противном случае — более короткое имя (рис. 4.20, *б*);

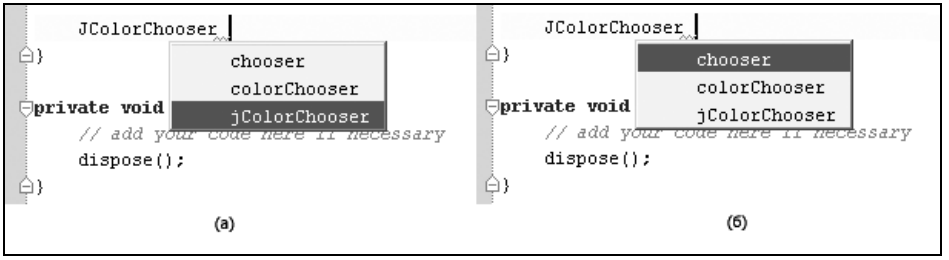


Рис. 4.20. Разные установки опции **Prefer longer names**: (*а*) — выбрать длинные имена, (*б*) — короткие

  - Опции настройки шаблона имени переменных, в которых можно указать префикс и суффикс имени переменной:
    - ◊ **Field** (Поле класса);
    - ◊ **Static field** (Статическое поле класса);
    - ◊ **Parameter** (Параметр вызова метода);
    - ◊ **Local variable** (Локальная переменная).
- **Keep When Reformatting** (Сохранять при переформатировании). Данная группа содержит перечень пунктов составляющих кода, для которых можно указать, сохранять ли их при переформатировании кода или нет:
  - **Line breaks** — разрывы строк;
  - **Comment at first column** — комментарий в начале строки;
  - **Simple methods in one line** — простые методы, записанные на одной строке;
  - **Simple blocks in one line** — простые блоки, записанные в одной строке;
  - **Control statement in one line** — управляющие конструкции (например, `if`, `for` и т. п.), записанные в одной строке.
- **Order of Members** (Порядок членов класса). Данная группа содержит список из четырех пунктов, порядок которых можно менять с помощью

кнопок **Move Up** (Поднять на позицию вверх), **Move Down** (Опустить на позицию вниз). В соответствии с этой последовательностью IntelliJ IDEA будет вставлять в код автоматически создаваемые символы:

- **Fields** (Поля класса);
- **Constructors** (Конструкторы);
- **Methods** (Методы);
- **Inner classes** (Внутренние классы).

❐ Дополнительные:

- Опции расстановки комментариев:
  - ◊ **Line comment at first column** — строчный комментарий в начале строки;
  - ◊ **Block comment at first column** — блочный комментарий в начале строки;
- **Line separator (for new files)** — разделитель строк (для новых строк):
  - ◊ **System Dependent** — в зависимости от системы;
  - ◊ **Unix \n** — как в Unix;
  - ◊ **Windows \r\n** — как в Windows;
  - ◊ **Mac \r** — как в Mac;
- **Right margin (columns)** — правая граница (в колонках).

4.1.7.2. Закладка *Indent and Braces* *(Отступ красной строки и фигурные скобки)*

Опции на данной закладке объединены в пять групп (рис. 4.21).

В правой части закладки расположено поле предварительного просмотра (**Preview**), в котором показано, как будет выглядеть Java-код для текущих настроек красной строки и фигурных скобок:

❐ **Tabs and Indents** (Табуляция и отступ красной строки). В данной группе есть возможность настроить, с помощью каких символов будет создаваться отступ для красной строки. Данные параметры можно сделать общими для всех типов файлов: **Java** (Java-файлы), **Jsp** (JSP-файлы), **XML** (XML-файлы), **Other** (Прочие файлы). Чтобы для каждого типа файлов сделать индивидуальные настройки, нужно выключить опцию **Use same settings for all file types** (Использовать одинаковые настройки для всех типов файлов). Существуют следующие опции:

- **Use Tab Character** (Использовать символ табуляции) — использовать для отступа символ табуляции. Если опция выключена, будут использованы пробелы;

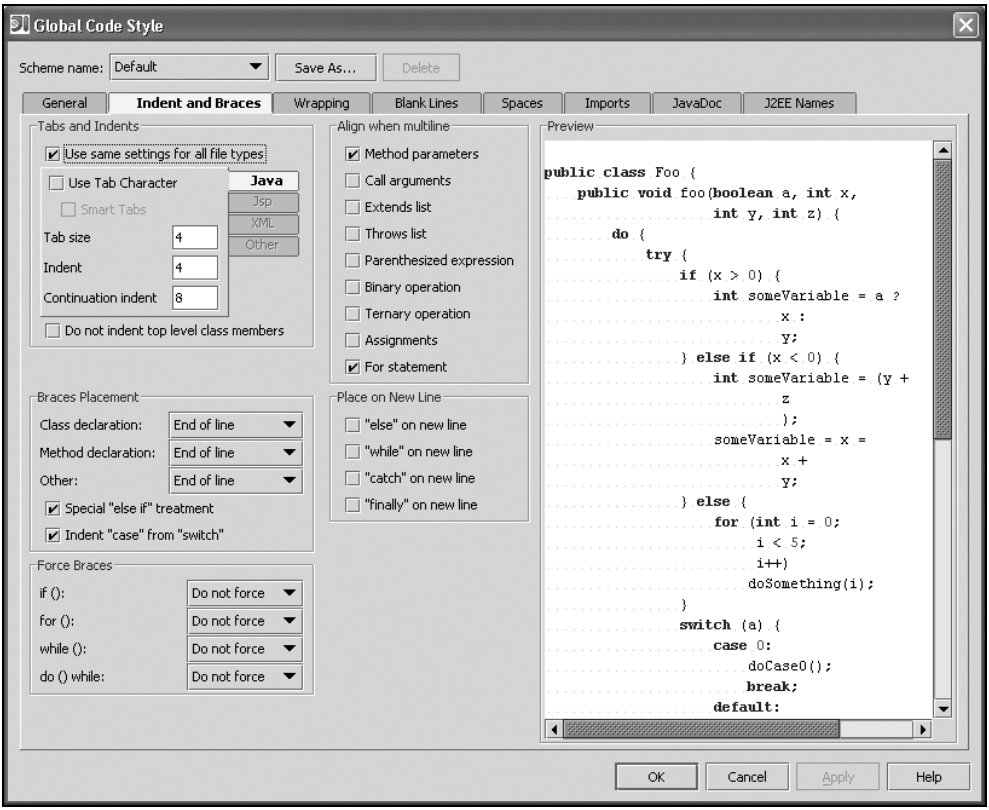


Рис. 4.21. Закладка **Indent and Braces** диалога **Global Code Style**

- **Smart Tabs** ("Умная" табуляция). Если "умная" табуляция включена, символы табуляции будут использованы только для указания отступа текущего контекста кода, а последующий отступ, если он требуется, будет задан с помощью пробелов (рис. 4.22). Данная опция может быть недоступна для изменения в следующих случаях:
  - ◊ выключена опция **Use Tab Character** (Использовать символ табуляции);
  - ◊ размер отступа (**Indent**) не кратен размеру табуляции (**Tab size**);

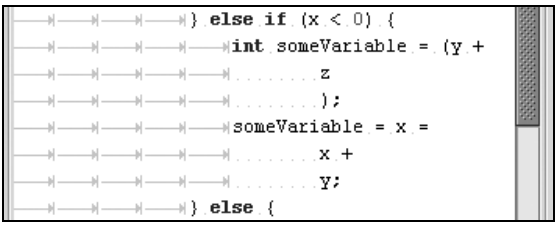


Рис. 4.22. "Умные" табуляции

- **Tab size** (Размер табуляции) — размер табуляции в символах;
- **Indent** (Отступ) — размер отступа в символах;
- **Continuation indent** (Продолжительный отступ) — размер отступа в символах, который IntelliJ IDEA вставит в случае переноса цельной конструкции кода на другую строку. Например, перенос инициализации переменной `someVariable` на рис. 4.22;
- **Do not indent top level class members** — не делать отступ для членов класса верхнего уровня.

❐ **Braces Placement** (Размещение фигурных скобок). Размещение фигурных скобок в коде. Для каждой из конструкций можно задать четыре возможных значения:

- **End of line** — скобку размещать в конце строки;
- **Next line** — на следующей строке без отступа;
- **Next line shifted** — на следующей строке с отступом от начала строки;
- **Next line shifted2** — на следующей строке с отступом от начала строки, а также со смещением вложенного контента.

Расположение скобок можно настроить для трех типов конструкций:

- **Class declaration** (Определение класса);
- **Method declaration** (Определение метода);
- **Other** (Прочие конструкции).

Также можно задать две дополнительные опции:

- **Special "else if" treatment** — специальная обработка конструкции `else if`. Если эта опция включена, между `else` и `if` не будет вставляться разрыв строки, а содержимое оператора `if` не будет иметь дополнительного отступа;
- **Indent "case" from "switch"** — отделять `case` от `switch` с помощью дополнительного отступа;

❐ **Force Braces** (Принудительные фигурные скобки). Опции из данной группы позволяют устанавливать фигурные скобки в принудительном порядке для следующих конструкций:

- **if ()**;
- **for ()**;
- **while ()**;
- **do () while**.

Можно задать три типа принудительной расстановки:

- **Do not force** — не делать принудительной расстановки;

- **When multiline** — делать принудительную расстановку, если конструкция содержит несколько строк кода;
- **Always** — всегда выполнять принудительную расстановку.

❐ **Align when multiple** (Выравнивать для нескольких строк). Данная группа объединяет опции, позволяющие задать дополнительное выравнивание конструкций кода, занимающих несколько строк:

- **Method parameters** — параметры определения метода;
- **Call arguments** — аргументы при вызове метода;
- **Extends list** — список наследования;
- **Throws list** — список выбрасываемых исключений;
- **Parenthesized expression** — выражение в скобках
- **Binary operation** — бинарная операция;
- **Ternary operation** — тернарная операция (`?:`);
- **Assignments** — операция присваивания;
- **For statement** — оператор `for`.

❐ **Place on New Line** (Размещать на новой строке). Группа опций, задающих возможность размещать конструкции кода на новой строке:

- **"else" on new line** — `else` на новой строке;
- **"while" on new line** — `while` на новой строке;
- **"catch" on new line** — `catch` на новой строке;
- **"finally" on new line** — `finally` на новой строке.

4.1.7.3. Закладка *Wrapping (Перенос)*

Опции переноса строк. В правой части данной закладки также находится окно предварительного просмотра изменений в настройках (**Preview**) (рис. 4.23).

В списке, расположенном в левой части закладки, перечислены все возможные опции переноса, для которых можно задать одно из следующих значений:

❐ **Do not wrap** — не переносить;

❐ **Wrap if long** — переносить, если строка слишком длинная и выходит за правую границу (*см. раздел 3.6.4*);

❐ **Chop down if long** — переносить, если строка слишком длинная и выходит за правую границу (*см. раздел 3.6.4*), располагая каждый перенесенный элемент на новой строке;

❐ **Wrap always** — переносить всегда.

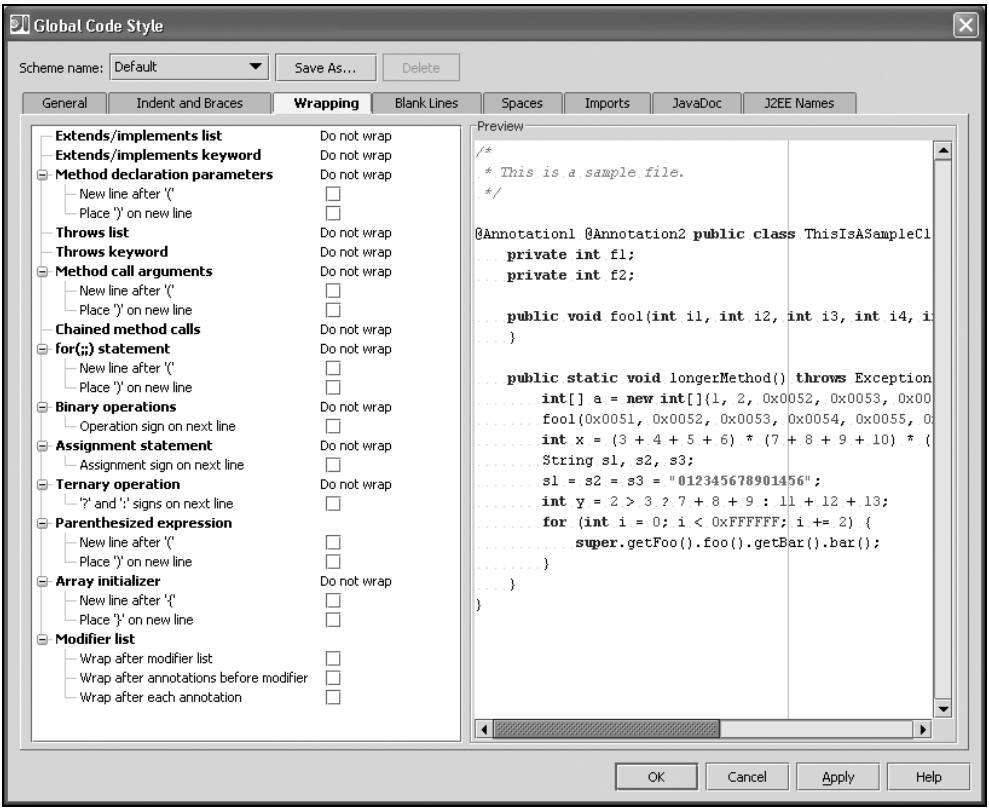


Рис. 4.23. Закладка **Wrapping** диалога **Global Code Style**

Вот эти опции:

- **Extends/implements list** — список наследования или имплементации;
- **Extends/implements keyword** — ключевое слово `extends` наследования или `implements` имплементации;
- **Method declaration parameters** — параметры определения метода. Для данной опции можно задать дополнительные свойства:
  - **New line after '('** — вставлять новую строку после `(`;
  - **Place ')' on new line** — размещать `)` на новой строке;
- **Throws list** — список выбрасываемых исключений;
- **Throws keyword** — ключевое слово списка `throws` выбрасываемых исключений;

- **Method call arguments** — аргументы вызова метода. Для данной опции можно задать дополнительные свойства:
  - **New line after '('** — вставлять новую строку после `(`;
  - **Place ')' on new line** — размещать `)` на новой строке;
- **Chained method calls** — вызовы методов по цепочке;
- **for(;;) statement** — оператор `for`. Для данной опции можно задать дополнительные свойства:
  - **New line after '('** — вставлять новую строку после `(`;
  - **Place ')' on new line** — размещать `)` на новой строке;
- **Binary operations** — бинарные операторы. Для данной опции можно задать дополнительное свойство **Operation sign on next line** (Размещать символ операции на новой строке);
- **Assignment statement** — оператор присваивания. Для данной опции также можно задать дополнительное свойство **Assignment sign on next line** — размещать символ оператора присваивания (`=`) на новой строке;
- **Ternary operation** — тернарный оператор. Данная опция имеет дополнительное свойство **'?' and ':' signs on next line** — размещать символы `?` и `:` на следующей строке;
- **Parenthesized expression** — выражение в круглых скобках. Для данной опции можно задать дополнительные свойства:
  - **New line after '('** — вставлять новую строку после `(`;
  - **Place ')' on new line** — размещать `)` на новой строке;
- **Array initializer** — инициализация массива. Данная опция имеет дополнительные свойства:
  - **New line after '{'** — вставлять новую строку после `{`;
  - **Place '}' on new line** — размещать `}` на новой строке;
- **Modifier list** — свойства списка модификаторов:
  - **Wrap after modifier list** — вставлять перенос после списка модификаторов;
  - **Wrap after annotations before modifier** — вставлять перенос после аннотаций перед списком модификаторов;
  - **Wrap after each annotation** — вставлять перенос после каждой аннотации.

4.1.7.4. Закладка *Blank lines (Пустые строки)*

Данная закладка содержит две группы опций настройки размещения пустых строк. В правой части закладки имеется панель предварительного просмотра настроек (**Preview**) (рис. 4.24).

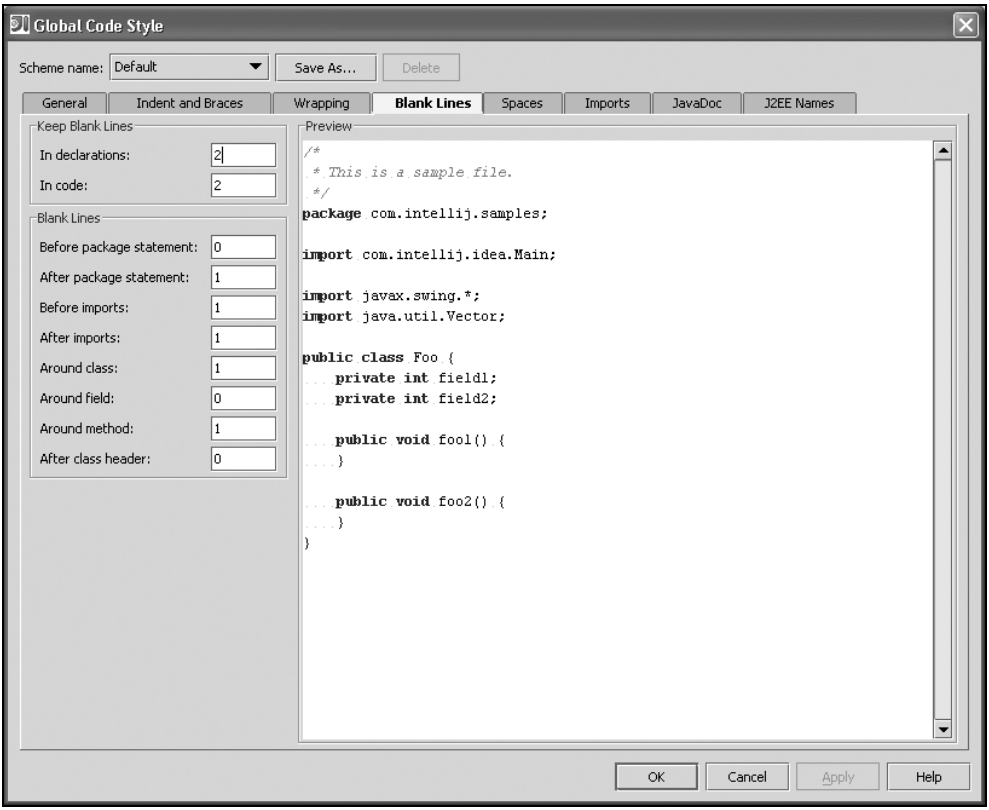


Рис. 4.24. Закладка **Blank Lines** диалога **Global Code Style**

Для каждой опции можно задать количество пустых строк. Если в конкретном случае пустые строки не нужны, следует указать 0.

- **Keep Blank Lines** — группа опций, задающая правила сохранения пустых строк в коде:
  - **In declarations** — количество пустых строк в определениях;
  - **In code** — количество пустых строк в коде;
- **Blank Lines** — группа опций настройки количества пустых строк:
  - **Before package statement** — перед определением пакета;
  - **After package statement** — после определения пакета;
  - **Before imports** — перед списком импортов;
  - **After imports** — после списка импортов;
  - **Around class** — вокруг класса;
  - **Around field** — вокруг поля;

- **Around method** — вокруг метода;
- **After class header** — после заголовка класса.

4.1.7.5. Закладка *Spaces (Пробелы)*

Данная закладка содержит целый список однотипных опций, задающих необходимость расстановки пробелов в том или ином случае (рис. 4.25).

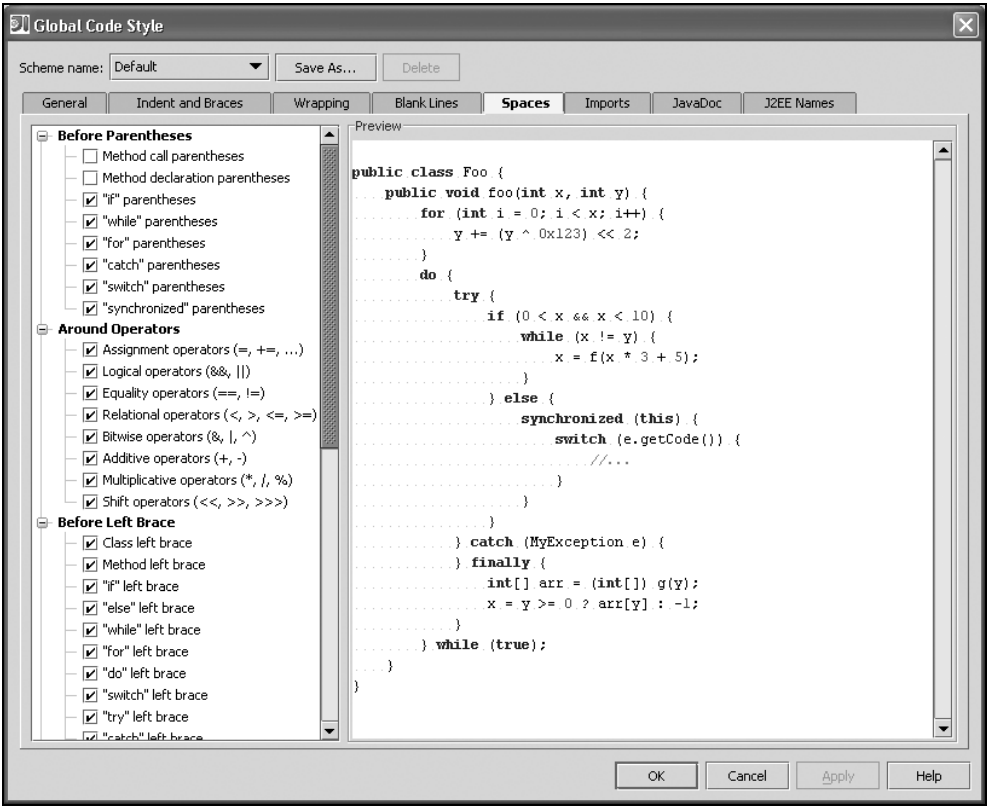


Рис. 4.25. Закладка **Spaces** диалога **Global Code Style**

Все опции разбиты на группы:

- **Before Parentheses** — пробел необходим перед круглыми скобками в случаях:
  - **Method call parentheses** — вызова метода;
  - **Method declaration parentheses** — при определении метода;
  - **"if" parentheses** — у оператора `if`;
  - **"while" parentheses** — у оператора `while`;

- **"for" parentheses** — у оператора `for`;
- **"catch" parentheses** — у оператора `catch`;
- **"switch" parentheses** — у оператора `switch`;
- **"synchronized" parentheses** — у оператора `synchronized`;

❐ **Around Operators** — пробел необходим вокруг (слева и справа):

- **Assignment operators (=, +=, ...)** — операторов присваивания;
- **Logical operators (&&, ||)** — логических операторов;
- **Equality operators (==, !=)** — операторов сравнения;
- **Relational operators (<, >, <=, >=)** — операторов сравнения;
- **Bitwise operators (&, |, ^)** — битовых операторов;
- **Additive operators (+, -)** — операторов сложения/вычитания;
- **Multiplicative operators (*, /, %)** — операторов умножения/деления;
- **Shift operators (<<, >>, >>>)** — операторов сдвига;

❐ **Before Left Brace** — пробел необходим перед левой фигурной скобкой:

- **Class left brace** — перед левой фигурной скобкой класса;
- **Method left brace** — перед левой фигурной скобкой метода;
- **"if" left brace** — перед левой фигурной скобкой оператора `if`;
- **"else" left brace** — перед левой фигурной скобкой оператора `else`;
- **"while" left brace** — перед левой фигурной скобкой оператора `while`;
- **"for" left brace** — перед левой фигурной скобкой оператора `for`;
- **"do" left brace** — перед левой фигурной скобкой оператора `do`;
- **"switch" left brace** — перед левой фигурной скобкой оператора `switch`;
- **"try" left brace** — перед левой фигурной скобкой оператора `try`;
- **"catch" left brace** — перед левой фигурной скобкой оператора `catch`;
- **"finally" left brace** — перед левой фигурной скобкой оператора `finally`;
- **"synchronized" left brace** — перед левой фигурной скобкой оператора `synchronized`;

❐ **Within Parenthesis** — пробел необходим внутри круглых скобок для:

- **Parenthesis** — выражения;
- **Method call parenthesis** — вызова метода;
- **Method declaration parenthesis** — определения метода;
- **"if" parenthesis** — условия оператора `if`;

- **"while" parenthesis** — условия оператора `while`;
- **"for" parenthesis** — параметров оператора `for`;
- **"catch" parenthesis** — параметра оператора `catch`;
- **"switch" parenthesis** — параметра оператора `switch`;
- **"synchronized" parenthesis** — параметра оператора `synchronized`;
- **Type cast parenthesis** — приведения типа;

❐ **In Ternary Operator (?:)** — пробел необходим в тернарном операторе:

- **Before '?'** — перед `?`;
- **After '?'** — после `?`;
- **Before ':'** — перед `:`;
- **After ':'** — после `:`;

❐ **Other** — прочие случаи расстановки пробела:

- **Within brackets** — внутри квадратных скобок;
- **After comma** — после запятой;
- **Before comma** — перед запятой;
- **After semicolon** — после точки с запятой;
- **Before semicolon** — перед точкой с запятой;
- **After type cast** — после приведения типа.

4.1.7.6. Закладка *Imports (Строки импорта)*

Данная закладка содержит настройки импортов (рис. 4.26).

Закладка **Imports** содержит три группы опций:

❐ **General** — общие настройки стиля импортов:

- **Use single class import** — создавать строки импорта для каждого класса в отдельности;
- **Use fully qualified class names** — использовать полные имена класса (с именем пакета);
- **Insert imports for inner classes** — включать строки импорта для внутренних классов;
- **Use fully qualified class names in javadoc** — использовать полные имена классов (с именем пакета) в JavaDoc;
- **Class count to use import with '*'** — использовать импорт с маской *, если количество классов из одного пакета превышает указанное количество;

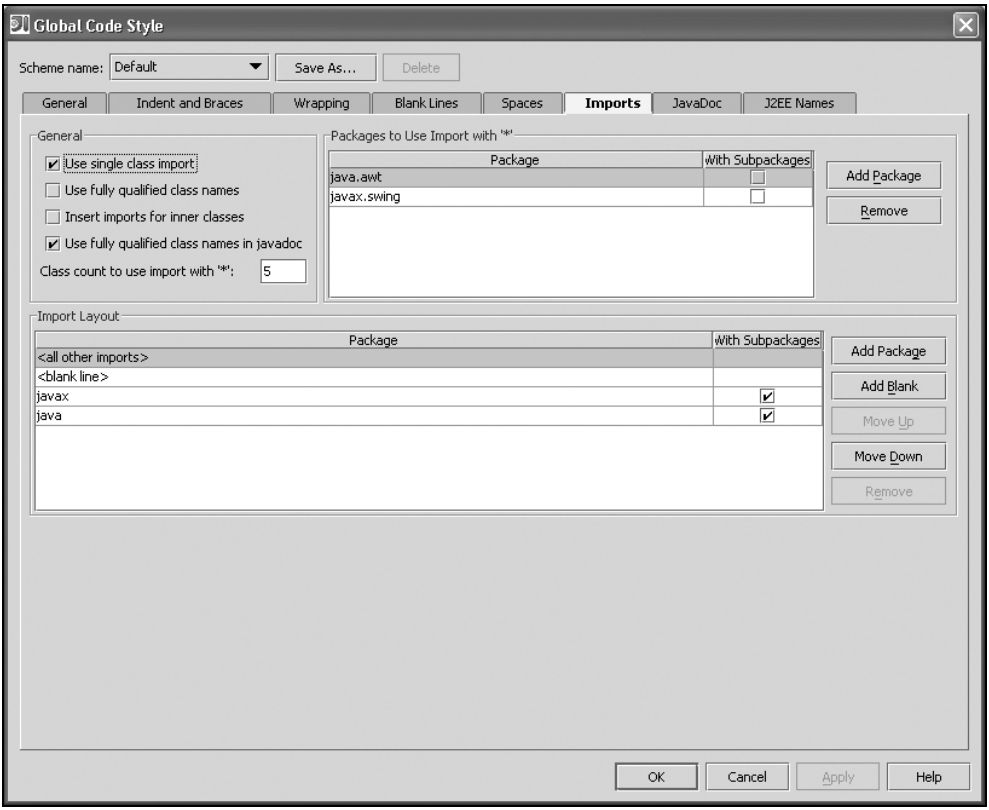


Рис. 4.26. Закладка **Imports** диалога **Global Code Style**

- **Packages to Use Import with '*'** — список пакетов, при импорте классов из которых надо использовать маску *. Добавлять и удалять пакеты можно с помощью двух кнопок: **Add Package** (Добавить пакет) и **Remove** (Удалить);
- **Import Layout** — настройка размещения строк импорта. В данном списке можно настроить порядок размещения импортов для разных групп пакетов, а также добавление пустой строки между этими группами. В списке имеется один постоянный пункт, который нельзя удалить, а можно только перемещать: **<all other imports>** (все остальные импорты). Этот пункт задает место размещения тех импортов, которые не попали под определения из списка. Для пакета из списка можно задать возможность включать все его вложенные пакеты. Так, на рис. 4.26 таких пакетов два: javax и java.

4.1.7.7. Закладка *JavaDoc (Комментарии JavaDoc)*

Данная закладка содержит опции настройки стиля комментариев JavaDoc (рис. 4.27).

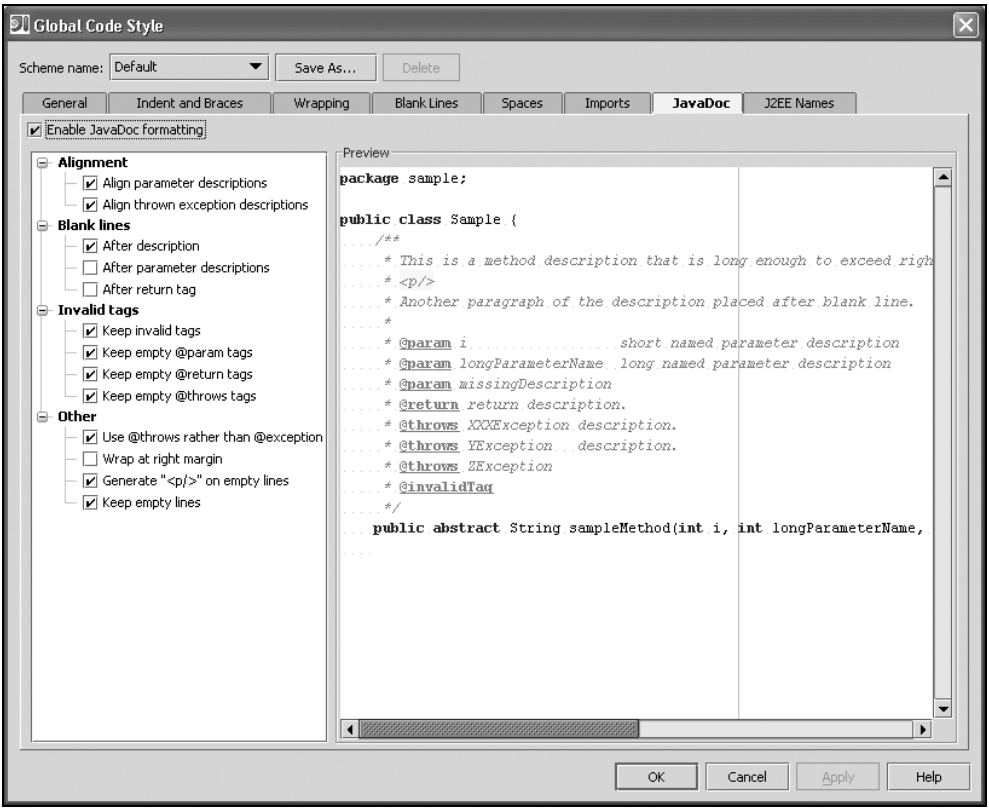


Рис. 4.27. Закладка **JavaDoc** диалога **Global Code Style**

В правой части закладки расположена панель предварительного просмотра настроек. Все опции сгруппированы и собраны в список, представленный в левой части закладки.

- **Alignment** — выравнивание:
  - **Align parameter description** — выравнивать описания параметров;
  - **Align thrown exception descriptions** — выравнивать описания выбрасываемых исключений.
- **Blank lines** — вставлять пустую строку:
  - **After description** — после описания;
  - **After parameter descriptions** — после описания параметра;
  - **After return tag** — после описания возвращаемого значения.
- **Invalid tags** — что делать с допустимыми тегами JavaDoc:
  - **Keep invalid tags** — сохранять недопустимые теги;
  - **Keep empty @param tags** — сохранять пустые теги `@param`;

- **Keep empty @return tags** — сохранять пустые теги `@return`;
- **Keep empty @throws tags** — сохранять пустые теги `@throws`.

❐ **Other** — прочие настройки стиля JavaDoc:

- **Use @throws rather than @exception** — использовать тег `@throws` вместо `@exception`;
- **Wrap at right margin** — переносить по правой границе (*см. раздел 4.1.3*);
- **Generate "<p/>" on empty lines** — вставлять "`<p/>`" в пустые строки комментария;
- **Keep empty lines** — сохранять пустые строки.

4.1.7.8. Закладка *J2EE Names (Имена J2EE)*

Данная закладка содержит настройки имен различных сущностей J2EE (рис. 4.28).

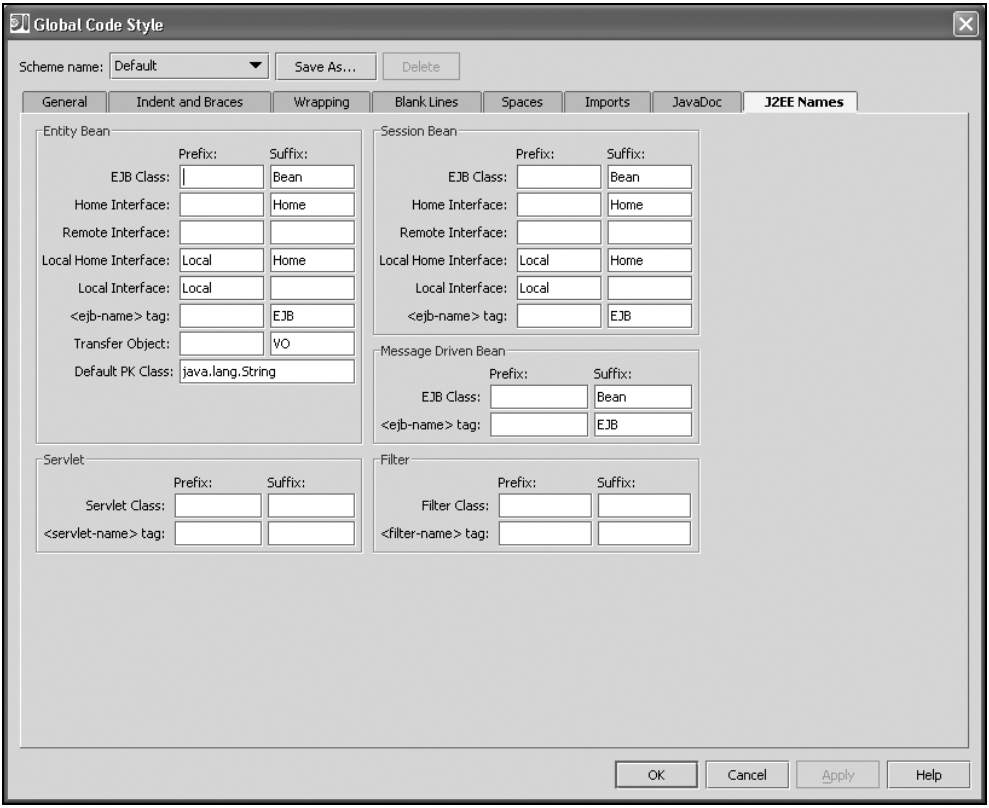


Рис. 4.28. Закладка **J2EE Names** диалога **Global Code Style**

Можно настроить префикс (**Prefix**) и суффикс (**Suffix**) имени следующих J2EE сущностей, а также их составляющих:

- **Entity Bean** (**EJB Class**, **Home Interface**, **Remote Interface**, **Local Home Interface**,**Local Interface**, **<ejb-name> tag**, **Transfer Object**, **Default PK Class**);
- **Session Bean** (**EJB Class**, **Home Interface**, **Remote Interface**, **Local Home Interface**, **Local Interface**, **<ejb-name> tag**);
- **Message Driven Bean** (**EJB Class**, **<ejb-name> tag**);
- **Servlet** (**Servlet Class**, **<servlet-name> tag**);
- **Filter** (**Filter Class**, **<filter-name> tag**).

4.1.8. Диалог настроек *JDK & Global Libraries (JDK и Глобальные библиотеки)*

Диалог настройки JDK и глобальных библиотек представлен на рис. 4.29.

Данный диалог содержит две закладки:

- **JDK** — с настройками JDK;
- **Global Libraries** — глобальные библиотеки.

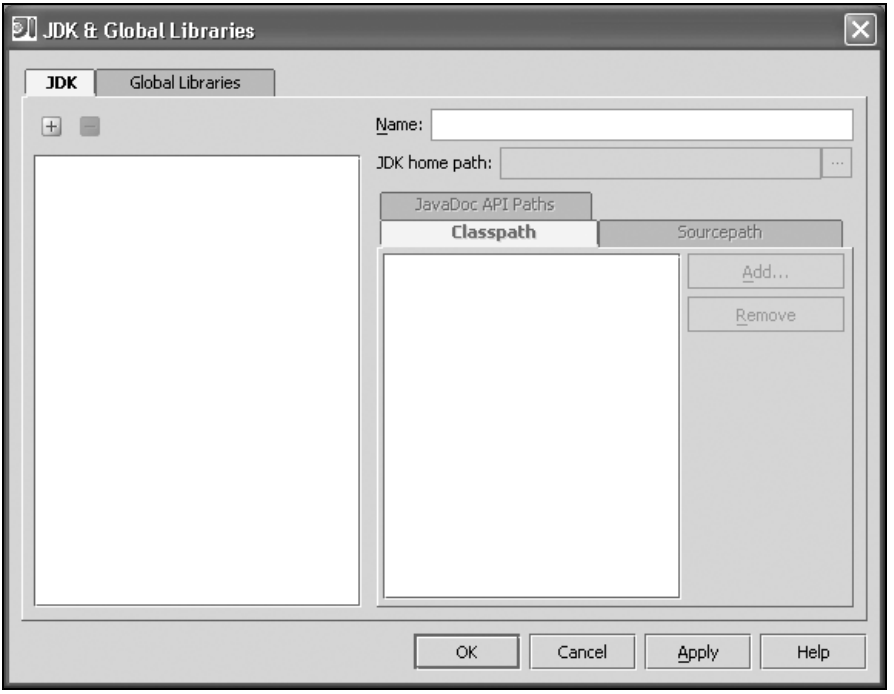


Рис. 4.29. Диалог настройки JDK и глобальных библиотек

4.1.8.1. Настройка JDK

Чтобы начать разработку любой Java-программы с помощью IntelliJ IDEA, необходимо настроить хотя бы один JDK. JRE, поставляемого в комплекте с IntelliJ IDEA, недостаточно, чтобы можно было компилировать программы.

Закладка с настройками JDK содержит следующие компоненты:

- список доступных JDK с возможностью добавления/удаления, расположенный в левой части;
- настройки конкретной версии JDK из списка.

Чтобы добавить JDK, нужно нажать соответствующую кнопку со знаком '+' и в появившемся диалоге выбрать директорию, в которой установлен JDK (рис. 4.30).

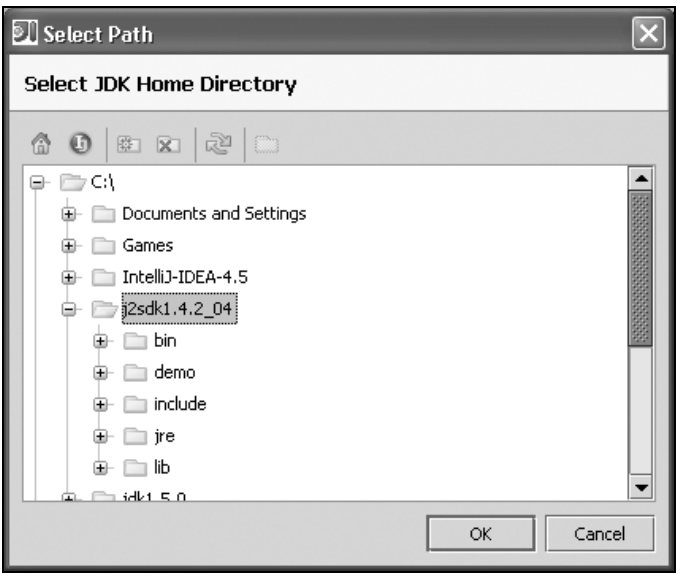


Рис. 4.30. Выбор директории с установленным JDK

Если директория была задана правильно, и IntelliJ IDEA смогла распознать JDK, установленный в ней, она автоматически настроит все его параметры (рис. 4.31).

У JDK есть следующие параметры:

- **Name** — название;
- **JDK home path** — домашний каталог JDK;
- **Classpath** — путь к классам JDK. IntelliJ IDEA учитывает все JAR-файлы библиотек JRE, за исключением файла tools.jar, который можно добавить самостоятельно, если необходимо;

- **Sourcepath** — путь к исходным текстам JDK;
- **JavaDoc API Paths** — пути к файлам комментариев JDK API.

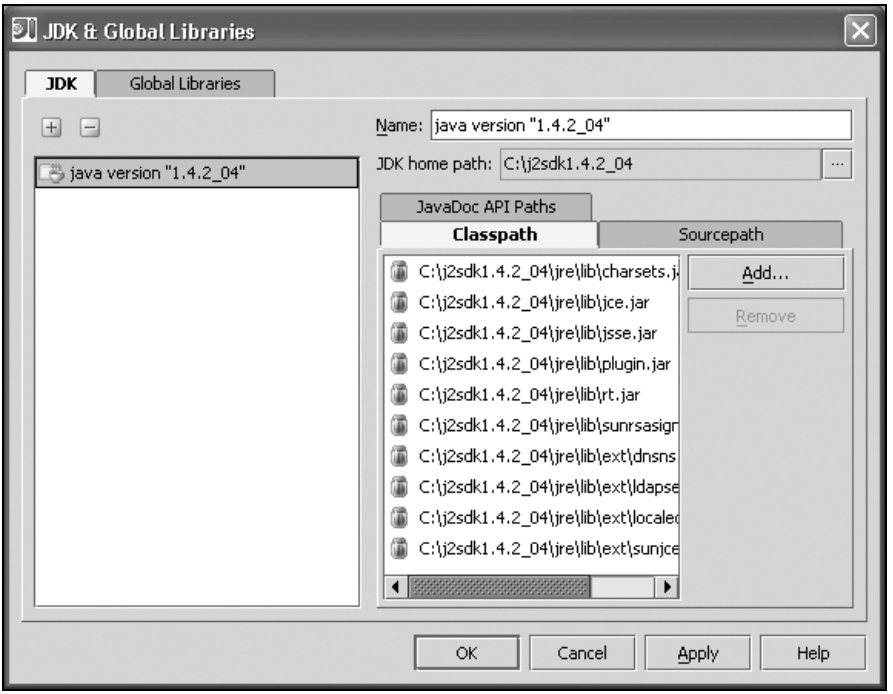


Рис. 4.31. JDK 1.4.2_04 добавлен

Три последних параметра располагаются на дополнительных закладках (см. рис. 4.31). В качестве пути к файлам комментариев JavaDoc API можно указать не только локальный путь, но и внешний адрес, где размещены файлы комментариев.

4.1.8.2. Настройка глобальных библиотек

Глобальные библиотеки (Global Libraries) — это библиотеки, которые можно использовать с любым проектом, разрабатываемым с помощью IntelliJ IDEA. По умолчанию список глобальных библиотек пуст (рис. 4.32).

Чтобы создать новую глобальную библиотеку, нужно нажать кнопку **Create Library** (Создать библиотеку) и выбрать директорию, в которой находятся файлы библиотеки, а также задать ее имя, если имя по умолчанию вас не устраивает (рис. 4.33).

После того, как библиотека создана, для нее можно задать список классов (в виде файлов JAR или ZIP), список исходных файлов, а также файлы или внешние ссылки комментариев JavaDoc (рис. 4.34).

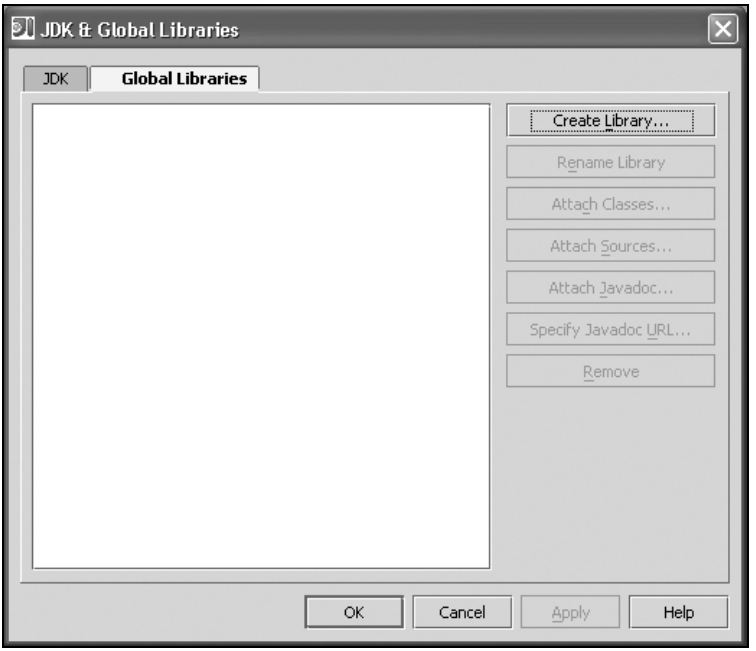


Рис. 4.32. Настройки глобальных библиотек

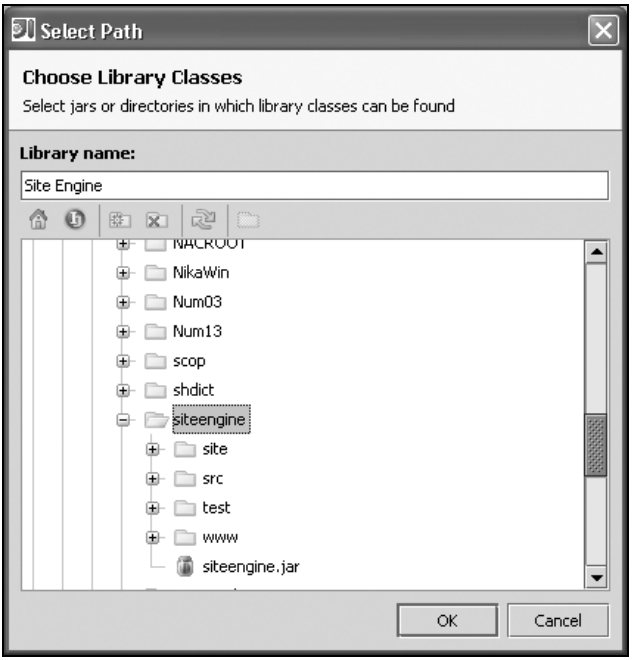


Рис. 4.33. Выбор директории глобальной библиотеки

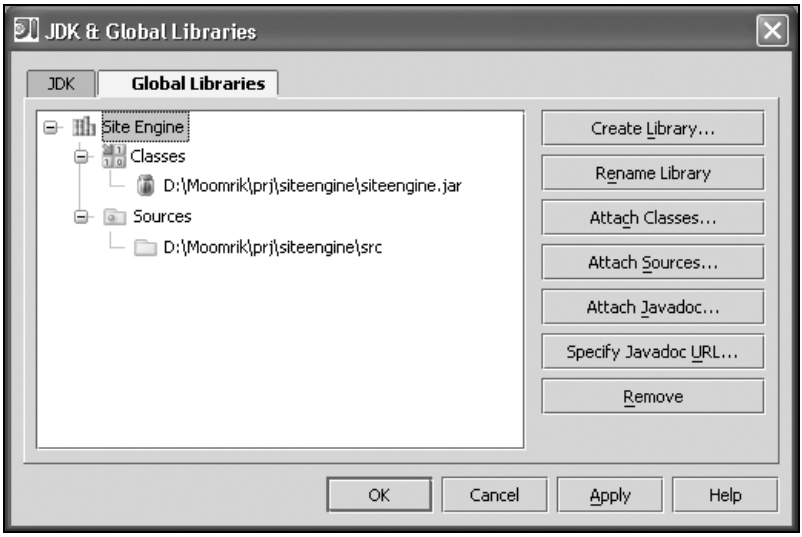


Рис. 4.34. Глобальная библиотека с добавленными классами и исходными текстами

Для добавления и изменения различных компонент глобальной библиотеки можно использовать соответствующие кнопки:

- **Rename Library** (Переименовать библиотеку);
- **Attach Classes** (Добавить классы);
- **Attach Sources** (Добавить исходные тексты);
- **Attach Javadoc** (Добавить файлы комментариев JavaDoc);
- **Specify Javadoc URL** (Указать адрес комментариев JavaDoc);
- **Remove** (Удалить компоненту библиотеки).

4.1.9. Диалог настроек *Debugger (Отладчик)*

Для настройки параметров отладчика используется диалог, изображенный на рис. 4.35.

Данный диалог имеет три группы опций:

- **General** (Общие настройки):
  - **Hide debug window on process termination** — скрывать инструментальное окно отладчика (*см. раздел 9.4*) при завершении отлаживаемого процесса;
  - **Reload classes after compilation** — перезагружать классы после компиляции (HotSwap):
    - **Always** (Всегда);
    - **Never** (Никогда);
    - **Ask** (По запросу);

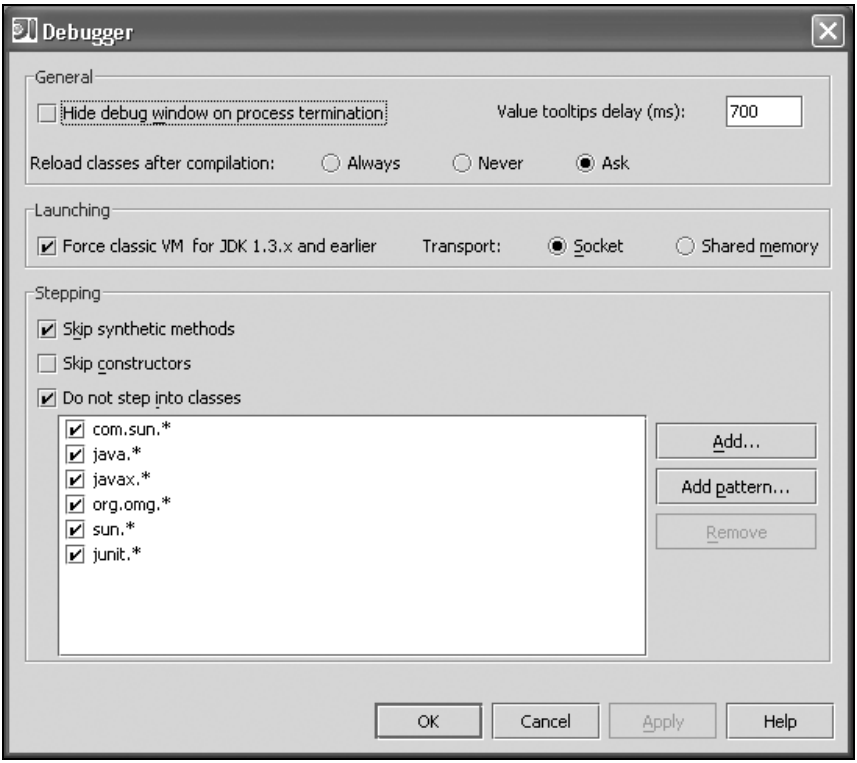


Рис. 4.35. Настройки отладчика

- **Value tooltips delay (ms)** — задержка появления всплывающих сообщений со значениями, возникающих при наведении курсора мыши на различные составляющие кода (переменные, поля класса и т. п.).

❒ **Launching** (Запуск) — запуск отладчика;

- **Force classic VM for JDK 1.3.x and earlier** — принудительно запускать классическую виртуальную машину для JDK версии 1.3.x и ниже;
- **Transport** — канал передачи данных, используемый при отладке:
  - ◊ **Socket** — сокет;
  - ◊ **Shared memory** — разделяемая память (данная опция доступна только при работе под Windows).

Примечание

Данные опции не имеют действия при удаленной отладке, за исключением случаев, когда отлаживаемое приложение запущено под IntelliJ IDEA.

- **Stepping** (Пошаговая отладка). Данные опции позволяют настроить пошаговую отладку:
  - **Skip synthetic methods** — пропускать синтетические методы — методы, созданные самим компилятором;
  - **Skip constructors** — пропускать вызовы конструкторов;
  - **Do not step into classes** — пропускать вызовы методов из классов, указанных в списке. В данном списке можно задать как конкретный класс (**Add**), так и набор классов через маску '*' (**Add pattern**). Синтаксис указания классов аналогичен синтаксису строк импорта.

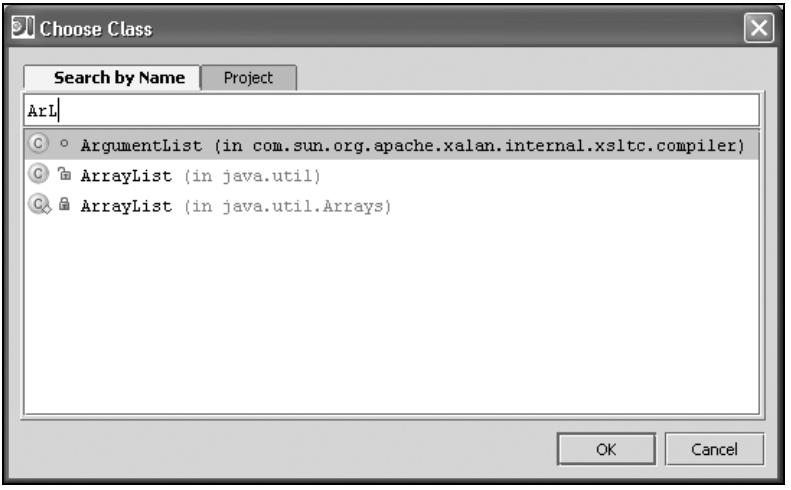


Рис. 4.36. Диалог выбора класса

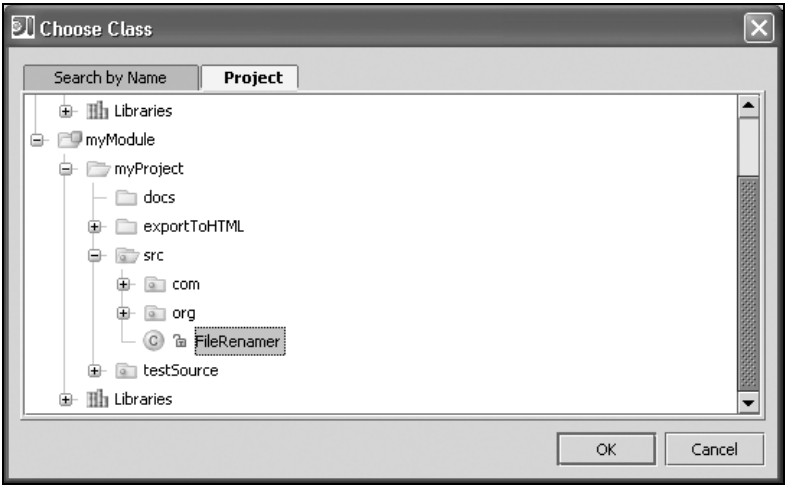


Рис. 4.37. Выбор класса из проекта

Для добавления классов используется специальный диалог выбора класса (рис. 4.36). В нем можно найти класс по имени, причем имеется возможность сокращенного именования, аналогичная той, что используется в команде выбора класса (*см. раздел 3.2.5.1*).

В качестве второй альтернативы можно выбрать класс из проекта (рис. 4.37).

4.1.10. Диалог настройки *Resources (Ресурсы)*

Диалог настройки ресурсов представлен на рис. 4.38.

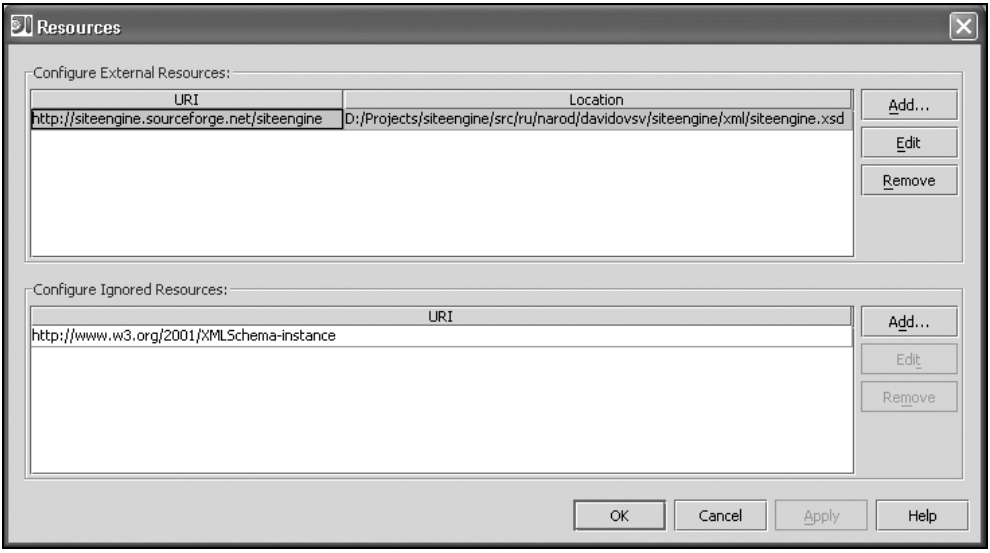


Рис. 4.38. Диалог настройки ресурсов

Под ресурсами понимаются URI-адреса, которые используются для спецификации типа XML-документов. С помощью диалога настроек ресурсов можно задать соответствие таких адресов файлам, локально сохраненным на диске, в которых эти ресурсы находятся (**Configure External Resources**).

Например, на рис. 4.38 задано соответствие для одного такого ресурса, содержащего XML-схему (XMLSchema). Когда в редакторе IntelliJ IDEA будет редактироваться XML-файл, использующий эту XML-схему, IntelliJ IDEA будет автоматически использовать содержимое указанного файла для валидации, подстановки типов, вывода информации об атрибутах и автоматического завершения кода.

С помощью диалога настройки ресурсов также можно указать игнорируемые ресурсы (**Configure Ignored Resources**), т. е. ресурсы, которые не будут использованы при работе с XML-документами.

4.1.11. Диалог настроек *File Types (Типы файлов)*

Диалог настройки типов файлов представлен на рис. 4.39.

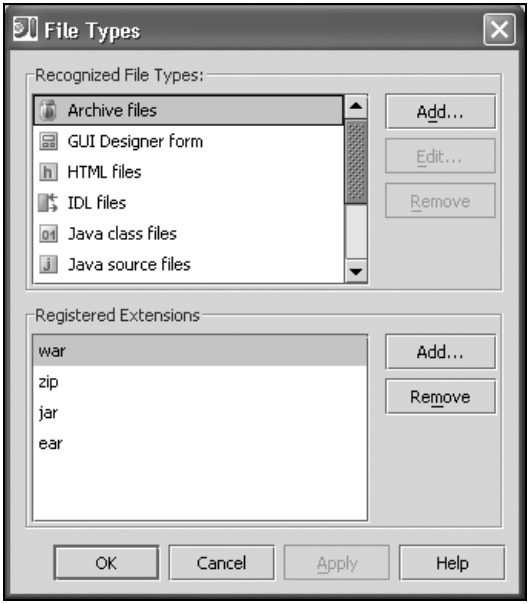


Рис. 4.39. Диалог **File Types**

Данный диалог предназначен для конфигурирования типов файлов, поддерживаемых IntelliJ IDEA. Тип файла распознается на основе его расширения. Каждому типу файлов может быть сопоставлено несколько расширений.

Изначально IntelliJ IDEA поддерживает следующие типы файлов (в скобках указаны расширения по умолчанию):

- **Archive files** — файлы архивов (war, zip, jar, ear);
- **GUI Designer form** — форма визуального редактора интерфейса пользователя IntelliJ IDEA (form);
- **HTML files** — HTML-файлы (html, shtml, htm, sht, shtm);
- **IDL files** — файлы описания интерфейса IDL — Interface Definition Library (idl);
- **Java class files** — файлы со скомпилированными Java-классами (class);
- **Java source files** — файлы с исходными текстами на Java (java);
- **JavaScript files** — файлы JavaScript (js);
- **JSP files** — JSP-файлы (jsp, jsf, jspf);

- **Text files** — текстовые файлы (sh, pl, cgi, log, properties, cmd, mf, sql, policy, txt, css, bat);
- **XML files** — XML-файлы (tld, xsd, xml, jnlp, xhtml, xsi, jhm, hs, dtd, wsdl).

Для некоторых встроенных типов файлов, например **IDL files** и **JavaScript files**, а также для всех пользовательских типов файлов, можно задать дополнительные настройки. В качестве примера создадим пользовательский тип файлов CMD files — пакетные файлы Windows XP (рис. 4.40).

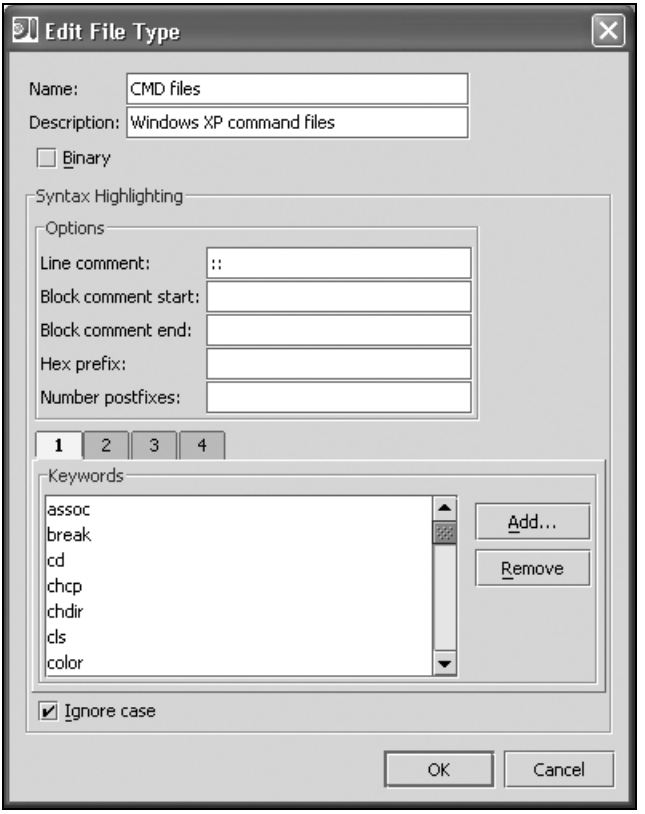


Рис. 4.40. Диалог редактирования свойств типа файлов

Создание и редактирование типа файлов производится с помощью диалога, представленного на рис. 4.40. С помощью этого диалога можно задать следующие опции:

- **Name** (Имя) — имя типа файлов;
- **Description** (Описание) — краткое описание типа файлов;
- **Binary** (Бинарный) — опция, используемая для указания типа файла. Если она выключена, файл считается текстовым;

- **Syntax Highlighting** (Подсветка синтаксиса) — в данной группе объединены опции, задающие настройки подсветки синтаксиса при редактировании файлов (*см. раздел 4.1.6.6*):
  - **Line comment** — символ строчного комментария (например, `//`);
  - **Block comment start** — начало блочного комментария, например, `/*`);
  - **Block comment end** — конец блочного комментария, например, `*/`);
  - **Hex prefix** — префикс шестнадцатеричных чисел, например, `0x`);
  - **Number postfixes** — окончания чисел, например, `L`, `F`;
  - **Keywords** (Ключевые слова) — ключевые слова разделены на четыре группы. В каждой группе задаются ключевые слова какого-то одного типа. Для каждого типа ключевых слов можно настроить стиль их представления в редакторе (*см. раздел 4.1.6.6*);
  - **Ignore case** (Не различать регистр символов).

В примере, представленном на рис. 4.40, ключевые слова CMD-файлов разделены на две группы. В первой группе представлены ключевые слова, поддерживаемые самим интерпретатором CMD.EXE, во второй группе — ключевые слова, поддерживаемые с помощью внешних запускаемых программ (в Windows XP они расположены в директории \Windows\system32). Такое деление не несет какой-либо дополнительной смысловой нагрузки и использовано просто в качестве примера.

После того как опции типа файлов настроены, необходимо задать расширения файлов (рис. 4.41).

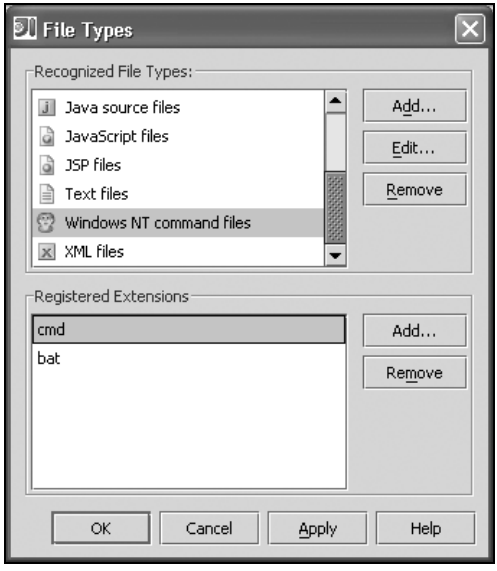


Рис. 4.41. Расширения файлов для пользовательского типа файлов

Если такие расширения уже зарегистрированы для каких-либо типов файлов (расширения из примера bat и cmd зарегистрированы на **Text files**), их придется сначала удалить из старых типов и уж потом добавить к новым.

В результате наших манипуляций, мы сможем пользоваться подсветкой синтаксиса для созданного нами типа файлов (рис. 4.42). Подробнее о настройках подсветки синтаксиса для таких файлов *см. раздел 4.1.6.6.*

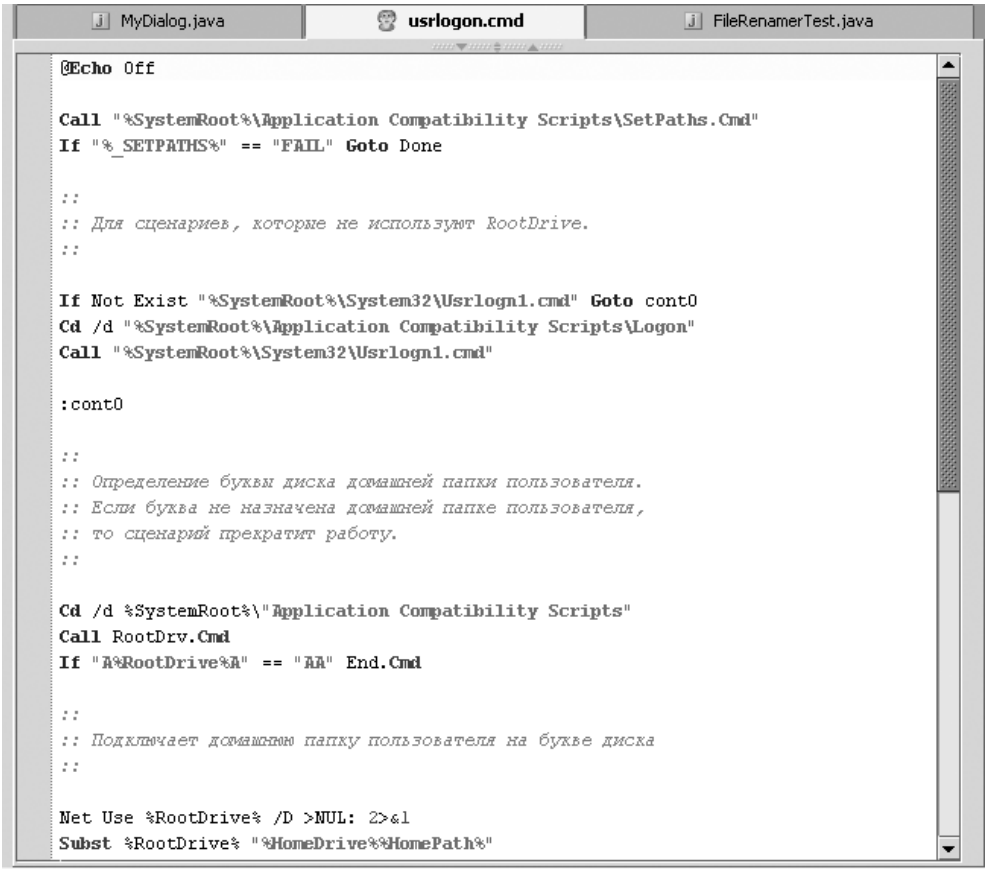


Рис. 4.42. Подсветка синтаксиса для пользовательского типа файла

Возможности подсветки синтаксиса пользовательских типов файлов невелики. Лучшего результата можно достичь лишь с помощью плагинов IntelliJ IDEA (*см. главу 15*).

4.1.12. Диалог настроек *Local History (Локальная история)*

Настройка локальной системы контроля версий IntelliJ IDEA производится с помощью диалога, представленного на рис. 4.43. Подробнее о локальной системе контроля версий и о работе с ней *см. раздел 12.2*.

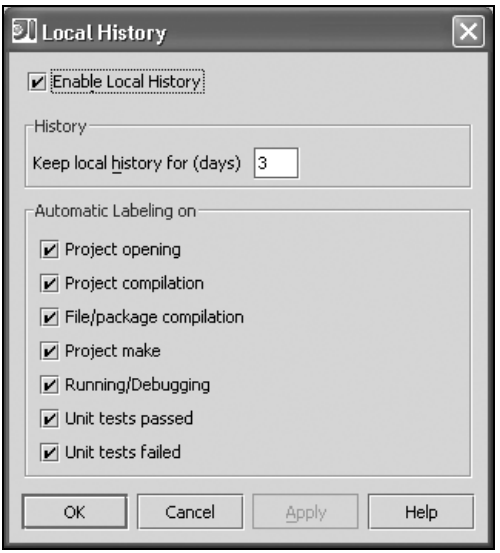


Рис. 4.43. Диалог настроек локальной системы контроля версий

В самом верху диалога расположена опция, позволяющая выключить поддержку локальной системы контроля версий. По умолчанию она включена, и мы бы очень не рекомендовали ее выключать.

Прочие опции данного диалога разделены на две группы:

- **History** (История). В данной группе содержится всего одна опция, задающая длительность хранения локальной истории в днях (**Keep local history for (days)**);
- **Automatic Labeling on** (Автоматическая расстановка меток) — автоматически помечать метками изменения при потенциально-опасных операциях, которые могут повредить файлы:
  - **Project opening** — открытии проекта;
  - **Project compilation** — компиляции проекта;
  - **File/package compilation** — компиляции отдельного файла/пакета;
  - **Project make** — сборке проекта;

- **Running/Debugging** — запуске/отладке;
- **Unit tests passed** — успешном прохождении Unit-тестов;
- **Unit tests failed** — ошибках в Unit-тестах.

4.1.13. *Live Templates ("Живые шаблоны")*

"Живые шаблоны" — одна из замечательных возможностей IntelliJ IDEA, существенно упрощающих написание кода.

Суть "живого шаблона" довольно проста — создается короткое символьное сокращение (аббревиатура), которому сопоставляются некоторые фрагменты кода, заданные в виде шаблона. Когда нужно написать код, который уже содержится в каком-либо "живом шаблоне", программисту требуется только набрать аббревиатуру и вызвать команды для раскрытия шаблона, который ей соответствует (*см. раздел 3.2.6.7*). Все остальное сделает за него IntelliJ IDEA.

Самым примечательным является то, что шаблон может содержать не просто статический текст, а заданный специальным образом код, который при подстановке в контекст программы может видоизменяться.

Приведем простой пример. Есть "живой шаблон", имеющий аббревиатуру "`itar`" — от сокращенного "iterate array" — итерировать массив. В листинге 4.2 приведен пример кода, к которому будет применен этот "живой шаблон". Все, что делает данная программа, — это выводит список аргументов, которые ей переданы при запуске.

Листинг 4.2

```
public class ITARExample {
    public static void main(String[] args) {
        itar
    }
}
```

Теперь если перевести курсор в конец строки с аббревиатурой "`itar`" и нажать <Ctrl>+<J> (вызов команды **Insert Live Template** (Вставить "живой шаблон"), *см. раздел 3.2.6.7*), то шаблон автоматически развернется в код, представленный в листинге 4.3.

Листинг 4.3

```
public class ITARExample {
    public static void main(String[] args) {
```

```
        for (int i = 0; i < args.length; i++) {
            String arg = args[i];


        }
    }
}
```

Все, что нам остается сделать, — это дописать вызов метода для вывода переменной `arg` в консоль.

После развертывания "живого шаблона", если у шаблона есть настраиваемые параметры (как это было в рассмотренном случае), IntelliJ IDEA позволяет уточнить отдельные составляющие результирующего кода (рис. 4.44).

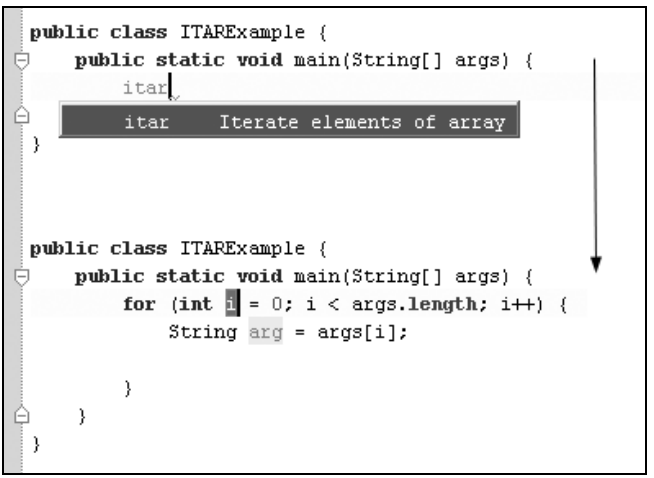


Рис. 4.44. Выбор значения дополнительных параметров шаблона

В данном случае мы могли выбрать название параметра цикла `i`, массив `args`, по которому хотим итерироваться, а также имя локальной переменной `arg`, которой присваивается значение из `i`-й ячейки массива.

Затем можно применить еще один "живой шаблон": "`soutv`" — System.out variable, который выводит в консоль значение переменной. Переместим курсор в пустую строку после объявления и инициализации переменной `arg`, напишем `soutv` и выберем переменную для печати (рис. 4.45).

Код готов. Мы потратили на него в несколько раз меньше времени, если бы писали его без помощи IntelliJ IDEA.

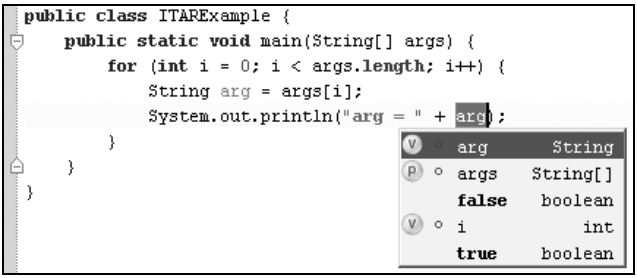


Рис. 4.45. Выбор переменной для печати в "живом шаблоне" `soutv`

4.1.13.1. Настройка "живых шаблонов"

"Живые шаблоны" — очень мощное средство, с помощью которого можно создавать код чрезвычайно быстро и качественно, поскольку возможность опечаток практически исключена.

Настройка "живых шаблонов" осуществляется с помощью диалога, представленного на рис. 3.46.

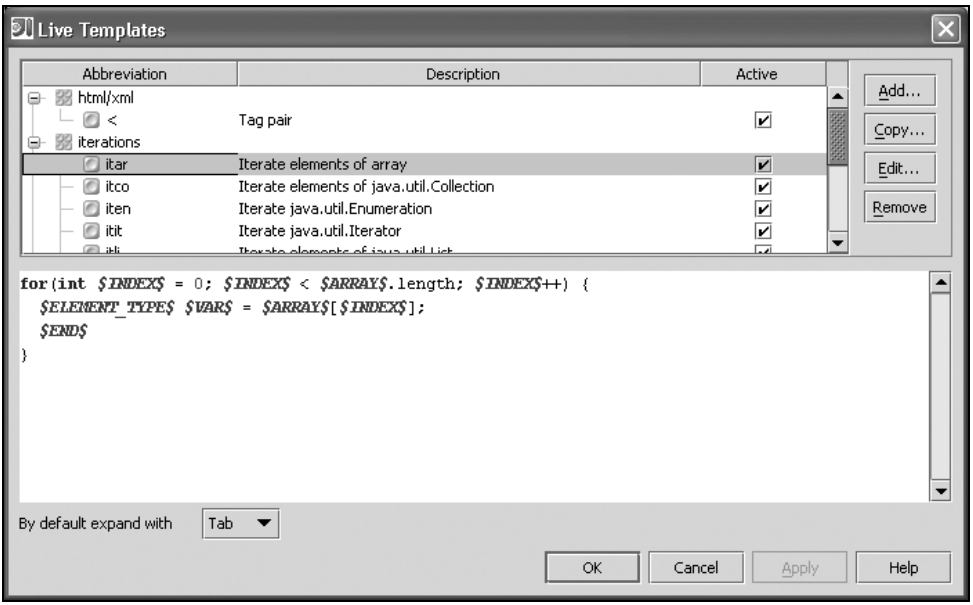


Рис. 4.46. Диалог настройки "живых шаблонов"

В IntelliJ IDEA есть набор из предопределенных шаблонов. Все шаблоны, в том числе и те, что программист может создать самостоятельно, разбиты на группы:

- **html/xml** — шаблоны для HTML и XML;
- **iterations** — шаблоны для итераций;

- **other** — прочие шаблоны;
- **output** — шаблоны вывода в консоль;
- **plain** — шаблоны сокращений простых кодовых конструкций;
- **surround** — шаблоны для обертывания (*см. раздел 3.2.6.8*);
- **user** — группа для шаблонов пользователя, не вошедших ни в одну из предыдущих.

Кроме указанных групп, можно создавать свои собственные.

Список "живых шаблонов" расположен в виде таблицы. В таблице содержится аббревиатура шаблона (**Abbreviation**), его описание (**Description**), а также опция для включения/выключения шаблона (**Active**).

С помощью опции **By default expand with** (По умолчанию раскрывать с) можно настроить клавишу, при нажатии которой шаблон будет активирован. Эта настройка действует по умолчанию для всех шаблонов, для которых не переопределена эта опция (*см. раздел 4.1.13.2*).

В нижней части диалога находится окно просмотра, в котором отображается текущий шаблон (рис. 4.46). По сути это та конструкция кода, в которую будет разворачиваться "живой шаблон".

Точная настройка "живого шаблона" производится с помощью диалога, представленного на рис. 4.47.

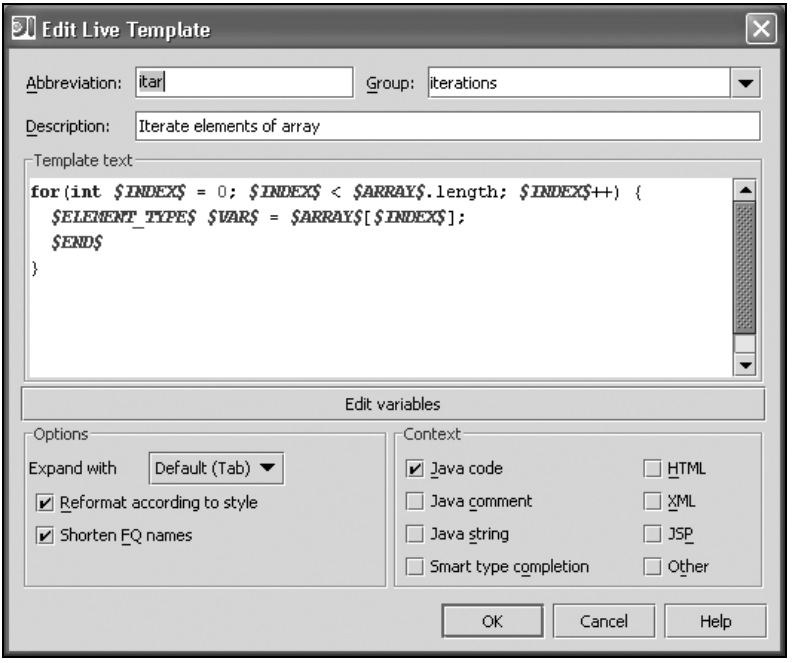


Рис. 4.47. Диалог редактирования "живого шаблона"

Данный диалог открывается, если создавать новый шаблон с помощью кнопки **Add** (Добавить), либо редактировать имеющийся с помощью кнопки **Edit** (Редактировать).

4.1.13.2. Редактирование шаблона

В диалоге редактирования "живого шаблона" есть следующие поля:

- **Abbreviation** (Аббревиатура) — короткое символьное сочетание, с помощью которого "живой шаблон" будет вызываться;
- **Description** (Описание) — краткое описание шаблона;
- **Group** (Группа) — группа, к которой относится шаблон. Можно выбрать одну из групп из выпадающего списка, можно создать новую, написав ее название в данном поле;
- **Template text** (Текст шаблона).

Также в диалоге редактирования шаблона имеются следующие опции, разбитые на две группы:

- **Options** (Опции) — общие опции шаблона:
  - **Expand with** (Раскрывать с) — индивидуальная клавиша раскрытия шаблона:
    - **Default** (Tab) — клавиша по умолчанию. Для всех шаблонов, у которых установлено значение опции **Expand with** (Раскрывать с) по умолчанию, используется значение данной опции из основного диалога настройки "живых шаблонов" (см. рис. 4.46);
    - **Space** — использовать <Space>;
    - **Tab** — использовать <Tab>;
    - **Enter** — использовать <Enter>;
  - **Reformat according to style** (Переформатировать согласно стилю). Данная опция позволяет переформатировать "живой шаблон" согласно стилю кода (*см. раздел 4.1.7*). В случае, если данная опция выключена, автоматически будут сделаны только отступы красной строки;
  - **Shorten FQ names** (Самые короткие FQ — Fully Qualified — полностью указанные имена) — в случае, если данная опция включена и в "живом шаблоне" использованы полные имена классов (с указанием пакета), IntelliJ IDEA будет использовать для их отображения максимально краткую запись (т. е. только имена классов), автоматически добавляя необходимые импорты.
- **Context** (Контекст) — контекст выполнения шаблона — типы файлов, в которых будет работать "живой шаблон":
  - **Java code** — Java-код;

- **Java comment** — комментарий в Java-файле;
- **Java string** — строка в Java-коде;
- **Smart type completion** — шаблон может участвовать в качестве варианта автоматического завершения кода по команде "умного" завершения кода (*см. раздел 6.1.5*);
- **HTML** — HTML-файлы;
- **XML** — XML-файлы;
- **JSP** — JSP-файлы;
- **Other** — прочие файлы.

4.1.13.3. Переменные

Кроме обычных кодовых или текстовых конструкций, в теле шаблона можно использовать *переменные шаблона*.

Предопределенные переменные

Имя переменной должно начинаться и заканчиваться знаком `$`. Существует две предопределенные переменные:

- `$END$` — позиция, в которой должен оказаться курсор, после развертывания "живого шаблона";
- `$SELECTION$` — переменная, используемая с шаблонами, предназначенными для "обертывания" с помощью команды **Surround With Live Template** (Обернуть "живым шаблоном") (<Ctrl>+<Alt>+<J>). Данная переменная задает содержимое блока выделения.

Пользовательские переменные

Все остальные переменные определяются пользователем. Настройка типов и значений переменных производится с помощью диалога, представленного на рис. 4.48.

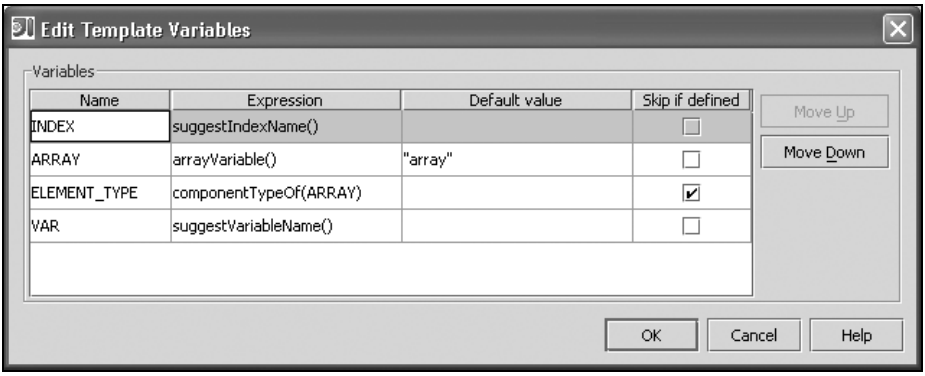


Рис. 4.48. Диалог настройки переменных "живого шаблона"

Данный диалог вызывается с помощью кнопки **Edit Variables** (Редактировать переменные) (см. рис. 4.47).

В данном диалоге содержится таблица с переменными, присутствующими в шаблоне. Переменные представлены в виде таблицы со следующими столбцами:

- **Name** (Имя) — имя переменной — символьная последовательность, записанная между знаками `$`;
- **Expression** — выражение переменной, которым данная переменная станет в момент активации "живого шаблона", т. е. ее функциональное значение;
- **Default value** — значение переменной по умолчанию. Данное значение будет использовано только в том случае, если для переменной задано некоторое выражение. В случае, если выражение не смогло вернуть никакого значения, будет использовано значение по умолчанию;
- **Skip if defined** — устанавливать значение переменной автоматически, если имеется только один вариант такого значения.

Последовательность переменных в таблице можно менять с помощью кнопок **Move Up** (Переместить вверх) и **Move Down** (Переместить вниз). Последовательность переменных в таблице совпадает с последовательностью обработки переменных в шаблоне.

Типы переменных

Переменные "живого шаблона" отличаются от переменных, например, в языке Java. В данном контексте переменной может являться не только переменная языка Java как таковая, но и тип или некоторое вспомогательное значение.

Основная "жизнь" "живого шаблона" как раз и заключается в этих переменных, вернее в их функциональном значении. В момент раскрытия "живого шаблона" переменные вычисляются в текущем контексте и заменяются некоторыми выражениями.

Раскрытие выражения может просто возвращать некоторое значение, а может выполнять некоторые дополнительные действия, например, предлагать выбрать некоторое значение из списка. То есть по своей сути эти выражения являются вызовами функций.

Рассмотрим выражения переменных подробнее. Далее мы будем говорить о них как о вызовах функций. Существует 20 типов выражений переменных "живого шаблона":

- `arrayVariable()` — предлагает все переменные типа массив из текущей области видимости;
- `capitalize(String)` — сделать заглавной первую букву имени, переданного в качестве параметра;

- `castToLeftSideType()` — приведение типа правой части выражения к типу левой части. Данная функция используется в "живых шаблонах" из группы итераций (iterations). Она создана для шаблонов, совместимых как с новыми возможностями языка Java версии 5, так и для старых версий. В Java 5 есть возможность использования параметризованных классов (generics), при этом необходимость в явном приведении типа отпадает. В старых версиях языка приведение типов необходимо;
- `className()` — возвращает имя класса, в контексте которого вызван "живой шаблон";
- `complete()` — вызвать команду **Code Completion** (Завершение кода) в текущей позиции (*см. раздел 6.1.1*);
- `completeSmart()` — вызвать команду **Smart Type Completion** ("Умное" завершение кода) в текущей позиции кода (*см. раздел 6.1.1*);
- `componentTypeOf(Array)` — функция возвращает тип элементов в массиве;
- `currentPackage()` — название пакета класса из текущего контекста;
- `decapitalize()` — сделать строчной первую букву имени, переданного в качестве параметра;
- `expectedType()` — функция возвращает тип, который ожидается в качестве результата работы всего шаблона. Работает, если шаблон выполняется для правой части присваивания, после оператора `return` и т. п.;
- `guessElementType(Container)` — функция возвращает предполагаемый тип элемента коллекции (`java.util.Collection`), переданной в качестве параметра. Чтобы сделать такое предположение, IntelliJ IDEA пытается найти места в коде, где элементы данной коллекции добавляются или извлекаются;
- `iterableComponentType(ArrayOrIterable)` — функция возвращает тип элемента итератора или массива;
- `iterableVariable()` — функция предлагает все итерируемые переменные (те, что имплементируют интерфейс `Iterable`, а также массивы);
- `lineNumber()` — функция возвращает номер текущей строки;
- `methodName()` — функция возвращает имя метода, в контексте которого был вызван "живой шаблон";
- `qualifiedClassName()` — функция возвращает полное имя класса, в контексте которого был вызван "живой шаблон". Чтобы данная функция отработала правильно, необходимо выключить опцию **Shorten FQ names** (Самые короткие FQ — Fully Qualified — полностью указанные имена) из настроек шаблона;
- `rightSideType()` — функция определяет тип левой части выражения в соответствии с его правой частью. Данная функция также создана для

написания "живых шаблонов", которые могут работать как с кодом, использующим новые возможности языка Java 5, так и с кодом, написанных на старых версиях Java;

- `suggestIndexName()` — функция предлагает имя индексной переменной. Если такой переменной еще нет в области видимости, функция возвращает `i`, в противном случае функция возвращает `j`, `k`, `l` и так далее — в зависимости от того, какой переменной первой не окажется в области видимости;
- `suggestVariableName()` — функция предлагает имя для переменной. Предложение основывается на типе переменной, а также на ее инициализирующем значении. Предлагаемые имена соответствуют настройкам стиля кода (*см. раздел 4.1.7*);
- `variableOfType(Type)` — функция предлагает все переменные, которые могут быть присвоены в качестве параметра указанного типа. Если в качестве параметра передается пустая строка, функция предложит все переменные вне зависимости от их типа.

4.1.13.4. Примеры использования

Далее мы подробно рассмотрим варианты использования каждого выражения для переменных на примере готовых "живых шаблонов".

Пример 1

Листинг 4.4

```
for(int $INDEX$ = 0; $INDEX$ < $ARRAY$.length; $INDEX$++) {
  $ELEMENT_TYPE$ $VAR$ = $ARRAY$[$INDEX$];
  $END$
}
```

В листинге 4.4 представлен текст "живого шаблона" `itar` — iterate array — итерация по массиву, который мы применяли к исходному коду из листинга 4.2 и получили в результате код из листинга 4.3. В данном шаблоне использовано четыре определяемых переменных со следующими выражениями:

- `INDEX` — `suggestIndexName()`;
- `ARRAY` — `arrayVariable()`;
- `ELEMENT_TYPE` — `componentTypeOf (ARRAY)`;
- `VAR` — `suggestVariableName()`.

Переменная `INDEX` — это переменная цикла, создаваемая с помощью выражения `suggestIndexName()`, которое подставляет в код имя переменной `i`.

Переменная ARRAY — это массив, элементы которого итерируются в цикле. Переменная имеет выражение arrayVariable(), которое находит все массивы в текущей области видимости и предлагает их на выбор для итерирования. В нашем случае (см. листинг 4.2) имеется только один массив args. Он и будет подставлен в код.

Переменная ELEMENT_TYPE имеет выражение componentTypeOf(ARRAY) — тип элемента массива, переданного в качестве параметра, т. е. массива, определяемого переменной ARRAY. Так как в нашем случае тип определяется однозначно (массив содержит элементы одного типа), тип элемента массива будет вычислен как String.

И последняя оставшаяся переменная VAR имеет выражение suggestVariableName() — предлагать имя переменной. В нашем случае будет предложено на выбор два значения: arg и s. Первое — по названию массива args, второе — по названию типа данных String.

Также в шаблоне используется встроенная переменная END, которая указывает, где будет располагаться курсор, после завершения развертывания "живого шаблона".

Пример 2

Листинг 4.5

```
for($ITER_TYPE$ $ITER$ = $COLLECTION$.iterator(); $ITER$.hasNext(); ) {
  $ELEMENT_TYPE$ $VAR$ =$CAST$ $ITER$.next();
  $END$
}
```

В листинге 4.5 представлен текст "живого шаблона" itco — iterate collection — итерация по коллекции. В данном шаблоне используются следующие переменные и выражения (переменные указаны в том порядке, в котором они перечислены в списке переменных шаблона):

- ITER — suggestVariableName();
- COLLECTION — variableOfType("java.util.Collection");
- ELEMENT_TYPE — guessElementType(COLLECTION);
- VAR — suggestVariableName();
- ITER_TYPE — rightSideType();
- CAST — castToLeftSideType().

Из тех выражений, которые мы еще не рассматривали, здесь используются rightSideType() и castToLeftSideType(). Прежде чем рассмотреть их ра-

боту, приведем два примера кода, получившихся после применения данного шаблона. В листинге 4.6 представлен код, использующий параметризованные классы (generics), в листинге 4.7 — не использующий.

Листинг 4.6

```
public static void main(String[] args) {
    Collection<String> argCollections =
              new ArrayList<String> (Arrays.asList(args));
    for (Iterator<String> iterator = argCollections.iterator();
                                     iterator.hasNext();) {
        String arg = iterator.next();

    }
}
```

Листинг 4.7

```
public static void main(String[] args) {
    Collection argCollections = new ArrayList (Arrays.asList(args));
    for (Iterator iterator = argCollections.iterator();
                            iterator.hasNext();) {
        // вариант 1
        Object o = iterator.next();
        // вариант 2
        String s = (String) iterator.next();

    }
}
```

Переменная `COLLECTION` определена выражением `variableOfType("java.util.Collection")`, которое находит все возможные переменные, подходящие по типу (в данном случае переменные, имплементирующие тип `java.util.Collection`).

Переменная `ELEMENT_TYPE` использует выражение `guessElementType(COLLECTION)`. В отличие от случая с шаблоном `itar` из листинга 4.4, мы не можем просто вычислить тип элементов коллекции, поэтому пытаемся его вычислить с помощью данного выражения. В случае кода, не использующего параметризованные классы из листинга 4.7, тип элемента коллекции был вычислен как `Object` (т. е. IntelliJ IDEA не смогла сделать предположение

о типе элемента в коллекции). Первый вариант кода из листинга 4.7 получен в результате выбора значений переменных "живого шаблона", предложенных IntelliJ IDEA. Во втором варианте мы помогли IntelliJ IDEA определить тип переменной `ELEMENT_TYPE`, заполнив его самостоятельно в редакторе. Переменная `CAST` во втором случае отработала иначе.

Для переменной `ITER_TYPE` использовано выражение `rightSideType()`, которое определяет тип, стоящий в правой части присваивания. В данном случае это выражение вычисляет тип выражения `$COLLECTION$.iterator()`, которое возвращает итератор коллекции, определенной переменной `COLLECTION`. В листинге 4.6 тип итератора определен как `Iterator<String>`, а в листинге 4.7 как `Iterator`. Именно так был вычислен тип выражения в правой части с использованием параметра класса и без него.

Переменная `CAST` определена как `castToLeftSideType()` — приведение выражения, стоящего в правой части, к типу в левой части. В случае листинга 4.6 данное приведение не требуется, благодаря использованию параметризованного класса (поскольку итератор `iterator` типизирован), а в листинге 4.7 приведение типов есть во втором варианте.

Рассмотренные выражения `suggestIndexName`, `arrayVariable`, `componentTypeOf`, `suggestVariableName`, `variableOfType`, `guessElementType`, `rightSideType`, `castToLeftSideType` используются в большинстве итеративных шаблонов.

Пример 3

Листинг 4.8

```
System.out.println("$CLASS_NAME$.$METHOD_NAME$");
```

В листинге 4.8 представлен шаблон `soutm` — System.out method, который выводит в консоль текущее имя класса и метода (что очень полезно для отладки). В данном шаблоне использованы следующие переменные:

- `CLASS_NAME` — `className()`;
- `METHOD_NAME` — `methodName()`.

Переменная `CLASS_NAME` использует выражение `className()`, возвращающее имя класса из текущего контекста.

Переменная `METHOD_NAME` использует выражение `methodName()`, возвращающее имя метода из текущего контекста.

Если данный шаблон применить к методу `main` класса `ITARExample` из листинга 4.3, получится код `System.out.println("ITARExample.main");`.

"Живые шаблоны", поставляемые вместе с IntelliJ IDEA, к сожалению, других выражений переменных не используют, что несколько усложняет понимание. Поэтому мы потрудились написать несколько собственных шаблонов, в которых использовали все "недостающие" выражения, а их не так уж мало: `capitalize`, `complete`, `completeSmart`, `currentPackage`, `decapitalize`, `expectedType`, `iterableComponentType`, `iterableVariable`, `lineNumber`, `qualifiedClassName`.

Пример 4

Рассмотрим пример, создающий `Logger` для класса. В листинге 4.9 представлен "живой шаблон" создания такой конструкции.

Листинг 4.9

```
public static final $CLASS_NAME$ LOG_$VAR_NAME$ =
                $CLASS_NAME$.getLogger("#$PACKAGE$.$MY_CLASS$");
```

Данный шаблон использует следующие переменные:

- `CLASS_NAME` — переменная, инициализированная только значением по умолчанию `Logger`. Так сделано для того, чтобы при необходимости можно было изменять это значение, не меняя самого текста шаблона;
- `VAR_NAME` — `decapitalize(MY_CLASS)` — преобразованное имя, полученное из значения переменной `MY_CLASS`, т. е. значение такой конструкции будет равно имени класса, записанного с маленькой буквы;
- `PACKAGE` — `currentPackage()` — текущий пакет;
- `MY_CLASS` — `className()` — имя текущего класса, для которого вызван "живой шаблон".

При вызове данного шаблона для класса `Circle`, находящегося в пакете com.mysite.myproject, будет сгенерирован код, представленный в листинге 4.10.

Листинг 4.10

```
public static final Logger LOG_circle =
                Logger.getLogger("#com.mysite.myproject.Circle");
```

Пример 5

В листинге 4.11 представлен текст "живого шаблона" `iter`, предназначенного для создания цикла итераций по массиву или коллекции с использованием новых возможностей цикла `for`, появившихся в Java 5 (*см. раздел 11.2*).

Листинг 4.11

```
for ($ELEMENT_TYPE$ $VAR$ : $ITERABLE_TYPE$) {
  $END$
}
```

Данный шаблон использует следующие переменные:

- `ITERABLE_TYPE` — `iterableVariable()` — переменная массива или коллекции (того типа, по которому может итерироваться новый цикл `for`) — итерабельного типа;
- `ELEMENT_TYPE` — `iterableComponentType(ITERABLE_TYPE)` — тип элементов итерабельного типа;
- `VAR` — `suggestVariableName()` — имя переменной цикла.

Для фрагмента кода из листинга 4.12 данный шаблон создает код, представленный в листинге 4.13.

Листинг 4.12

```
    public Parted(List<T> parts) {
        this.parts = parts;

    }
```

Листинг 4.13

```
    public Parted(List<T> parts) {
        this.parts = parts;
        for (T t : parts) {

        }
    }
```

4.1.14. Диалог настройки *File Templates (Шаблоны файлов)*

Настройка шаблонов файлов производится с помощью диалога, представленного на рис. 4.49.

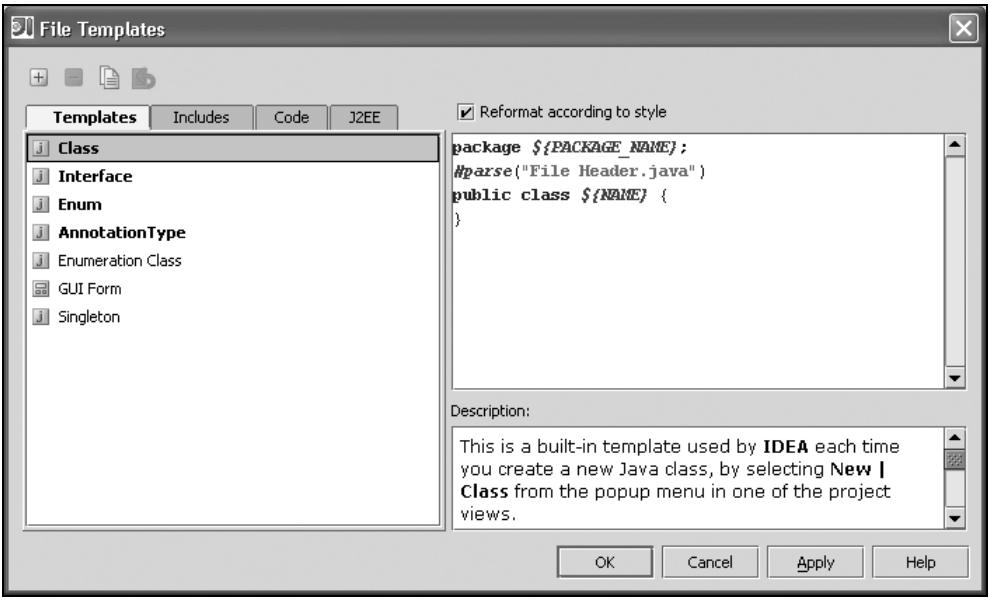


Рис. 4.49. Диалог настройки шаблонов файлов

Шаблоны файлов предназначены для быстрого создания однотипных файлов, а также частей кода стандартного вида.

Диалог состоит из трех частей: списка доступных шаблонов файлов, поля для редактирования текста шаблона, а также панели с описанием (**Description**).

С помощью кнопок на панели инструментов диалога шаблоны можно создавать, удалять, копировать, а также восстанавливать содержимое шаблонов (весьма полезная кнопка ☺).

Шаблоны файлов разделены на четыре группы, каждой из которых соответствует своя закладка в списке шаблонов:

- **Templates** (Шаблоны) — шаблоны создания файлов;
- **Includes** (Вставки) — шаблонные вставки, используемые первой группой шаблонов;
- **Code** (Код) — шаблоны кода;
- **J2EE** — шаблоны J2EE-файлов и конструкций.

Часть шаблонов выделена жирным шрифтом (см. рис. 4.49). Эти шаблоны можно изменить, но нельзя удалить, мы их будем называть *встроенными шаблонами* (*build-in*), чтобы отличать от прочих.

Шаблоны каждой группы имеют собственные переменные, с помощью которых их можно параметризировать. Некоторые предопределенные шаблоны имеют собственные переменные, которые можно использовать только в них.

Шаблоны файлов обрабатываются с помощью библиотеки Jakarta Velocity, подробная информация о которой доступна на сайте проекта Jakarta: **http://jakarta.apache.org/velocity/user-guide.html.**

Кроме имен переменных в шаблонах можно использовать условные выражения, циклы и все прочие возможности, предоставляемые библиотекой Velocity.

Имена переменных могут быть записаны следующим образом (где `VAR_NAME` — имя переменной):

- `$VAR_NAME`;
- `${VAR_NAME}`.

4.1.14.1. Закладка *Templates (Шаблоны)*

Данная закладка содержит шаблоны, предназначенные для создания однотипных файлов. Всего имеется семь предопределенных шаблонов, первые четыре из которых являются встроенными:

- **Class** (Класс). Шаблон, используемый при создании класса;
- **Interface** (Интерфейс). Шаблон, используемый при создании интерфейса;
- **Enum** (Перечисление). Шаблон, используемый при создании перечисления;
- **AnnotationType** (Аннотация). Шаблон, используемый при создании аннотации;
- **Enumeration Class** (Класс с перечислением). Шаблон класса — аналога перечисления (`Enum`) в Java 5, только реализованный средствами JDK 1.4 (*см. раздел 3.2.2.10*);
- **GUI Form** (Форма пользовательского интерфейса). Шаблон формы пользовательского интерфейса IntelliJ IDEA;
- **Singleton** (Синглтон). Шаблон класса, который может иметь только один экземпляр.

Данный перечень шаблонов доступен при вызове команды **New** (Создать) (*см. раздел 3.2.2.10*).

Шаблоны данной группы могут использовать следующие предопределенные переменные:

- `PACKAGE_NAME` — название пакета, в котором создаются файлы с помощью шаблонов данной группы;
- `USER` — логин данного пользователя;
- `DATE` — системная дата;
- `TIME` — системное время.

Встроенные шаблоны данной группы (**Class**, **Interface**, **Enum**, **AnnotationType**) могут дополнительно использовать переменную `NAME`, которая обозначает имя класса, интерфейса или перечисления.

Также для шаблонов данной группы можно использовать специальную инструкцию, которая позволяет включить в текст шаблона другой шаблон. В листинге 4.14 эта инструкция записана в первой строке: `#parse("File Header.java")`. Имя файла берется из шаблонов с закладки **Includes** (Вставки).

Кроме предопределенных, можно использовать переменные с другими именами. В этом случае IntelliJ IDEA включит поля ввода для них в диалог создания файла по шаблону.

Листинг 4.14

```
package ${PACKAGE_NAME};

public class ${Interface_Name}Impl implements ${Interface_Name} {

}
```

В листинге 4.14 представлен пример шаблона файла. В этом шаблоне используется предопределенная переменная `PACKAGE_NAME`, а также собственная переменная `Interface_Name`. Если использовать данный шаблон для создания файла, IntelliJ IDEA выведет диалог создания файла, представленный на рис. 4.50.

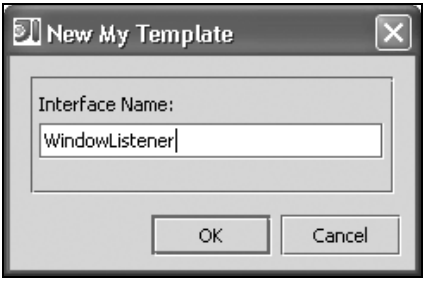


Рис. 4.50. Создание файла с помощью пользовательского шаблона

Как видно на рис. 4.50, IntelliJ IDEA заметила неизвестную ей переменную `Interface_Name` и отобразила для данной переменной поле ввода, снабдив его текстовым описанием, полученным из имени переменной путем замены знака _ (подчеркивание) на пробел. Название интерфейса `WindowListener` мы ввели сами. После подтверждения диалога был создан файл WindowListenerImpl.java, содержимое которого представлено в листинге 4.15.

Листинг 4.15

```
package com.mysite.myproject;

public class WindowListenerImpl implements WindowListener {

}
```

4.1.14.2. Закладка *Includes (Вставки)*

Шаблоны данной группы предназначены для вставки внутрь шаблонов из первой группы (**Templates**). Изначально в IntelliJ IDEA имеется один такой шаблон **File Header**. Данный шаблон является системным и используется в качестве заголовка классов, интерфейсов, перечислений и аннотаций, создаваемых с помощью соответствующих встроенных шаблонов **Class**, **Interface**, **Enum**, **AnnotationType**.

Поскольку шаблоны данной группы используются в качестве составной части шаблонами первой группы, они могут использовать все переменные, доступные шаблонам первой группы, а именно `PACKAGE_NAME`, `USER`, `DATE`, `TIME`.

> **Примечание**
>
> Несмотря на то, что в комментарии к шаблонам данной группы явно не указано на такую возможность, тем не менее, она есть: можно также использовать переменную `NAME` в качестве имени класса. Однако корректно работать такой шаблон из группы **Includes** (Вставки) будет только со встроенными шаблонами первой группы **Templates** (Шаблоны).

С помощью шаблонов-вставок удобно задавать различные типы "шапок" с комментариями к классам, информацию об авторских правах и пр.

4.1.14.3. Закладка *Code (Код)*

Данная группа содержит четыре встроенных шаблона, предназначенных для строго определенных целей:

- **Catch Statement Body** (Тело оператора `catch`) — шаблон тела оператора `catch`, генерируемый при вызове команды **Surround with** (Обернуть в) (<Ctrl>+<Alt>+<T>). Данный шаблон может использовать единственную предопределенную переменную `EXCEPTION`, обозначающую имя переменной-исключения, определенной в `catch`-блоке;
- **Implemented Method Body** (Тело имплементированного метода) — шаблон, используемый при генерации метода класса, имплементирующего

метод какого-либо интерфейса, с помощью команды **Implement Method** (Имплементировать метод) (<Ctrl>+<I>);

- **New Method Body** (Тело нового метода) — данный шаблон используется при автоматическом создании метода;
- **Overridden Method Body** (Тело переопределенного метода) — шаблон используется для генерации тела метода, переопределяющего метод из базового класса, с помощью команды **Override Method** (Переопределить метод). Данный шаблон имеет уникальную предопределенную переменную `CALL_SUPER`, которая создает вызов метода из базового класса. Если переопределяемый метод не возвращает значение, будет сгенерирован код вида `super.<имя метода>()`. Если метод возвращает значение, будет сгенерировано `return super.<имя метода>()`.

Данные шаблоны можно только модифицировать. Создать новый шаблон или удалить имеющийся невозможно.

Все шаблоны, кроме первого, могут использовать следующие предопределенные переменные:

- `RETURN_TYPE` — тип значения, возвращаемого методом;
- `DEFAULT_RETURN_VALUE` — значение, возвращаемое методом по умолчанию;
- `METHOD_NAME` — имя метода;
- `CLASS_NAME` — полное имя класса, для которого определен метод.

4.1.14.4. Закладка *J2EE*

Данная группа содержит шаблоны, используемые IntelliJ IDEA для создания J2EE-сущностей, таких как EJB, дескрипторы, сервлеты и т. п.

Все шаблоны данной группы также являются встроенными и их нельзя удалять или создавать новые. Можно только модифицировать текст имеющихся.

Так как J2EE-шаблонов довольно много, они представлены в виде иерархического списка со следующими разделами:

- **EJB** — шаблоны для создания EJB (Enterprise Java Bean):
  - **Java code templates** — шаблоны для Java-кода:
    - **Entity Bean Class BMP.java** — bean-managed persistence Entity Bean;
    - **Entity Bean Class CMP 1x.java** — container-managed persistence version 1.x Entity Bean;
    - **Entity Bean Class CMP 2x.java** — container-managed persistence version 2.x Entity Bean;
    - **Entity Home Interface.java** — Entity Bean remote home interface;
    - **Entity Local Home Interface.java** — Entity Bean local home interface;
    - **EJB Local Interface.java** — EJB local interface;

- ◊ **EJB Remote Interface.java** — EJB remote interface;
- ◊ **Session Bean Class Stateful.java** — Stateful Session Bean;
- ◊ **Session Bean Class Stateless.java** — Stateless Session Bean;
- ◊ **Sesion Home Interface.java** — Session Bean remote home interface;
- ◊ **Sesion Local Home Interface.java** — Session Bean local home interface;
- ◊ **Message Driven Bean Class.java** — Message Driven Bean;

- **Deployment descriptors**:
  - ◊ **ejb-jar.1_1.xml** — version 1.x ejb-jar.xml;
  - ◊ **ejb-jar.2_0.xml** — version 2.x ejb-jar.xml;

❐ **Application**:

- **Deployment descriptors**:
  - ◊ **application.1_2.xml** — application.xml версии 1.2;
  - ◊ **application.1_3.xml** — application.xml версии 1.3;
  - ◊ **application.1_4.xml** — application.xml версии 1.4;

❐ **Web**:

- **Java code templates** — шаблоны Java-кода:
  - ◊ **Servlet Class.java** — шаблон для создания сервлета;
  - ◊ **Filter Class.java** — шаблон для создания фильтра;
- **Deployment descriptors**:
  - ◊ **web.2_2.xml** — web.xml for Servlet 2.2 specification;
  - ◊ **web.2_3.xml** — web.xml for Servlet 2.3 specification;
  - ◊ **web.2_4.xml** — web.xml for Servlet 2.4 specification;
- **Jsp files** — шаблоны JSP-файлов:
  - ◊ **Jsp File.jsp** — шаблон JSP-файла;

❐ **WebLogic** — шаблоны для сервера приложений WebLogic:

- **Version 6.x** — шаблоны для WebLogic 6.x:
  - ◊ **weblogic-application.6x.xml** — WebLogic 6.x specific application deployment descriptor (weblogic-application.xml);
  - ◊ **weblogic.6x.xml** — web application;
  - ◊ **weblogic-cmp-rdbms.6x.xml** — weblogic-cmp-rdbms.xml;
  - ◊ **weblogic-ejb-jar.6x.xml** — weblogic-ejb-jar.xml;
- **Version 7.x** — шаблоны для WebLogic 7.x:
  - ◊ **weblogic-application.7x.xml** — WebLogic 7.x specific application deployment descriptor (weblogic-application.xml);

- ◊ **weblogic.7x.xml**;
- ◊ **weblogic-cmp-rdbms.7x.xml**;
- ◊ **weblogic-ejb-jar.7x.xml**;

- **Version 8.x** — шаблоны для WebLogic 8.x:
  - ◊ **weblogic-application.8x.xml** — WebLogic 8.x specific application deployment descriptor (weblogic-application.xml);
  - ◊ **weblogic.8x.xml**;
  - ◊ **weblogic-cmp-rdbms.8x.xml**;
  - ◊ **weblogic-ejb-jar.8x.xml**.

4.1.15. Диалог настройки *Keymap (Сочетания клавиш)*

Сочетания клавиш вызова команд IntelliJ IDEA настраиваются с помощью диалога, представленного на рис. 4.51.

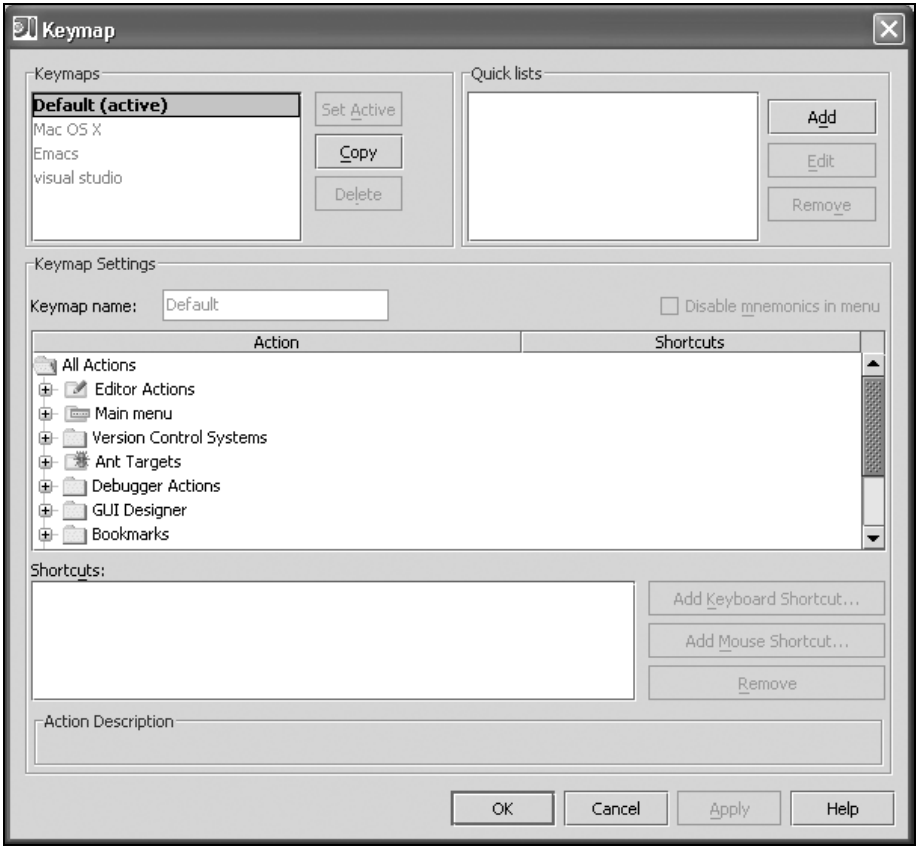


Рис. 4.51. Диалог настроек "горячих" клавиш

Настройки сочетания клавиш объединены в схемы, перечисленные на панели **Keymaps** (Сочетания клавиш), между которыми можно быстро переключаться (*см. раздел 3.2.4.15*).

4.1.15.1. Предопределенные схемы

В IntelliJ IDEA имеется четыре предопределенных схемы сочетания клавиш:

- **Default** — схема по умолчанию;
- **Mac OS X** — схема, использующая стандартные сочетания клавиш из Mac OS X;
- **Emacs** — схема, используемая в популярном редакторе Emacs;
- **visual studio** — схема клавиш, используемая в Microsoft Visual Studio.

Схема, активная в данный момент, отмечена жирным шрифтом и словом **Default** (по умолчанию). Выбрать активную схему можно с помощью кнопки **Set Active** (Сделать активной).

Предопределенные схемы не могут быть удалены, однако они могут быть использованы для создания собственных схем (с помощью кнопки **Copy** (Копировать)). Пользовательские схемы могут быть удалены с помощью кнопки **Delete** (Удалить).

4.1.15.2. Создание собственной схемы

Если требуется настроить сочетания клавиш, сначала нужно скопировать имеющуюся схему в свою собственную, и уже потом изменять настройки.

Все команды IntelliJ IDEA собраны в иерархическом списке, расположенном на панели **Keymap Settings** (Настройки сочетания клавиш) (см. рис. 4.51).

На данной панели также расположено поле **Keymap name** (Имя схемы сочетания клавиш), в котором можно просматривать имя схемы, настраиваемой в данный момент. Если это пользовательская схема, с помощью данного поля ее имя можно изменить.

Также на данной панели расположена опция **Disable mnemonics in menu** (Отключить мнемоники в меню). Мнемоники — это сочетания клавиш <Alt>+<буква>, подчеркнутая в названии пункта меню (см. рис. 3.1). Необходимость отключить мнемоники может пригодиться, если вы захотите использовать их сочетания клавиш в своих целях.

В низу панели **Keymap Settings** (Настройки сочетания клавиш) содержится две дополнительные панели:

- **Shortcuts** ("Горячие" клавиши). В этой панели представлен список "горячих" клавиш для вызова команды. У команды может быть несколько сочетаний "горячих" клавиш на клавиатуре плюс несколько сочетаний,

использующих кнопки мыши с возможностью одновременного нажатия клавиатурных клавиш. Сочетания можно добавлять с помощью кнопок:

- **Add Keyboard Shortcut** — добавить клавиатурное сочетание "горячих" клавиш;
- **Add Mouse Shortcut** — добавить сочетание "горячих" клавиш, вызываемое с помощью кнопок мыши.

Также "горячие" клавиши можно "отсоединять" от команды с помощью кнопки **Remove** (Удалить);

- **Action Description** (Описания команды). На данной панели выводится описание команды, если оно у команды есть. Это описание также выводится в строке статуса (*см. раздел 3.5*), когда вы перемещаетесь по пунктам главного меню (*см. раздел 3.2*).

Создание и редактирование "горячих" клавиш производится с помощью диалога, представленного на рис. 4.52.

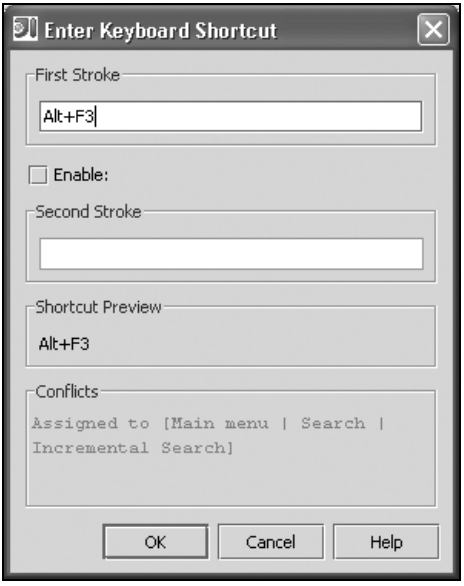


Рис. 4.52. Диалог создания и редактирования "горячих" клавиш

Данный диалог имеет четыре настройки:

- **First Stroke** (Первое сочетание) — первое сочетание клавиш — поле для ввода, отображающее нажатые в данный момент клавиши;
- **Second Stroke** (Второе сочетание) — дополнительное сочетание клавиш. Чтобы ввести дополнительное сочетание клавиш, необходимо включить поле для их ввода с помощью опции **Enable** (Разрешить);

- **Shortcut Preview** (Предварительный просмотр сочетания) — панель предварительного просмотра сочетаний клавиш. В сущности, оно повторяет то, что записано в полях **First Stroke** (Первое сочетание) и **Second Stroke** (Второе сочетание);
- **Conflicts** (Конфликты) — панель, на которой выводится сообщение в случае, если редактируемое сочетание клавиш для команды вступает в конфликт с сочетанием клавиш другой команды.

Если было задано и первое, и второе сочетание клавиш, вызов команды выглядит следующим образом:

- сначала нужно нажать первое сочетание клавиш;
- затем нажать второе.

Редактирование сочетания клавиш с использованием кнопок мыши производится с помощью диалога, представленного на рис. 4.53.

Данный диалог отличается от клавиатурного диалога дополнительной панелью **Click Pad** (Место для щелканья мышью), которую можно использовать для ввода "мышиных" "горячих" клавиш.

Рис. 4.53. Диалог создания и редактирования "горячих" клавиш с использованием кнопок мыши

4.1.15.3. Команды

Как мы уже говорили, все команды IntelliJ IDEA, а их немало (в главном меню, *см. раздел 3.2*, представлены далеко не все команды), сгруппированы в иерархический список. Далее мы приведем краткий список групп.

Editor Actions (Команды редактора)

В данной группе содержатся все команды, которые можно выполнить в редакторе IntelliJ IDEA (*см. раздел 3.6.4*).

Main Menu (Главное меню)

Полный список команд из главного меню.

Version Control Systems (Системы контроля версий)

Тут содержатся команды систем контроля версий. IntelliJ IDEA имеет встроенную поддержку трех систем контроля версий:

- SourceSafe. Поддержка на уровне собственной функциональности IntelliJ IDEA;
- CVS. Поддерживается с помощью плагина;
- Starteam. Поддерживается с помощью плагина.

Ant Targets (Пункты выполнения утилиты Ant[1])

Группа команд, представленных в файлах сборки утилиты Ant, настроенных для проекта.

Debugger Actions (Команды отладчика)

Команды отладчика IntelliJ IDEA.

GUI Designer (Команды визуального редактора пользовательского интерфейса)

Команды визуального редактора пользовательского интерфейса IntelliJ IDEA.

Bookmarks (Команды закладок)

Команды установки и перехода по закладкам (*см. раздел 6.2.8*).

External Tools (Команды внешних утилит)

Команды, соответствующие внешним утилитам, настроенным в IntelliJ IDEA (*см. раздел 4.1.16*). Команды объединены в группы, которые можно настроить для внешних утилит.

Macros (Команды макрокоманд)

Команды вызова макрокоманд (*см. раздел 3.2.11.7*).

Quick Lists (Команды из "быстрых" списков)

Команды, собранные в списки быстрого вызова. Подробнее об этих списках будет рассказано в конце этого раздела.

Other (Прочие команды)

Прочие команды, не вошедшие в предыдущие группы.

---

[1] Сначала нам очень хотелось привести данный пункт в дословном переводе — "Муравьиные цели", который красиво звучит, однако совершенно не выражает сути происходящего ☺.

4.1.15.4. Списки быстрого вызова

Списки быстрого вызова (**Quick Lists**) — весьма удобное средство, значительно упрощающее вызов команд. Суть списков быстрого вызова проста — во-первых, позволить пользователю создавать собственные наборы команд (аналог всплывающих меню). Команды можно вызывать с помощью "горячих" клавиш, однако в IntelliJ IDEA такое количество команд, что "горячих" клавиш на всех не хватит, да и запутаться в сотне команд проще простого.

Панель со списками быстрого вызова располагается в правой верхней части диалога настроек (см. рис. 4.51). Списки можно добавлять с помощью кнопки **Add** (Добавить), редактировать с помощью кнопки **Edit** (Редактировать) и удалять с помощью кнопки **Delete** (Удалить).

Создание и редактирование производится с помощью диалога, представленного на рис. 4.54.

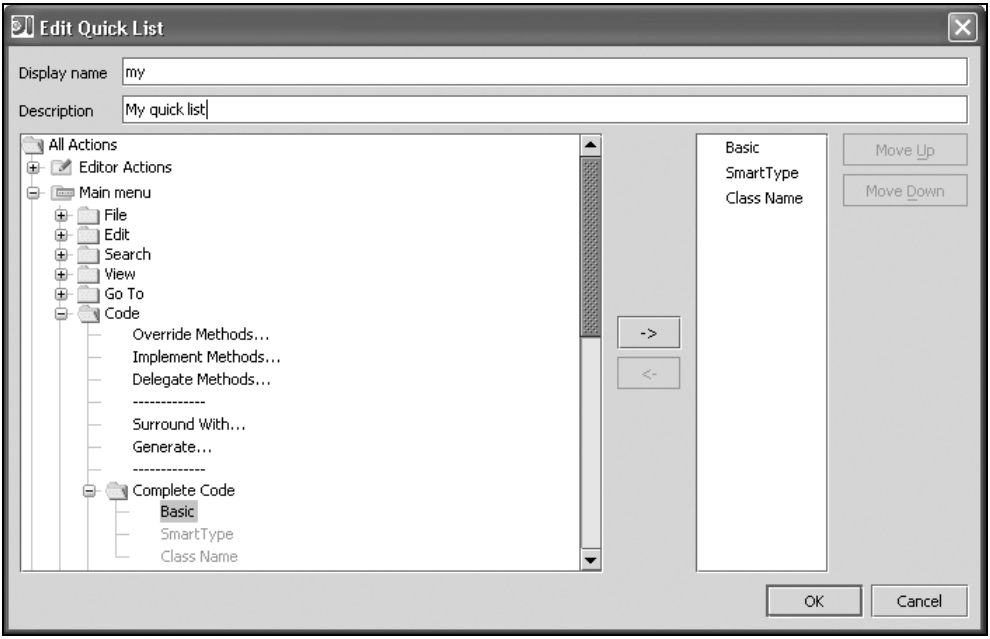


Рис. 4.54. Диалог редактирования списков быстрого вызова

В данном диалоге можно задать:

- название списка быстрого вызова (поле **Display name**);
- его описание (поле **Description**);
- а также команды из общего списка команд IntelliJ IDEA, которые будут входить в список быстрого вызова.

Составлять список очень просто: достаточно в списке команд найти необходимую, нажать кнопку со стрелкой вправо "->", и команда окажется в списке. Точно так же команды из списка можно удалять.

С помощью кнопок **Move Up** (Переместить вверх) и **Move Down** (Переместить вниз) можно менять последовательность команд в списке быстрого вызова.

Когда список быстрого вызова создан, ему можно назначить "горячие" клавиши так, как и для любой другой команды (рис. 4.55).

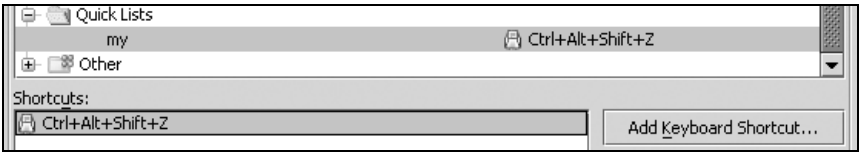


Рис. 4.55. Назначение "горячих" клавиш для вызова списка

Теперь, если нажать сочетание клавиш <Ctrl>+<Alt>+<Shift>+<Z>, появится всплывающее меню с командами из списка быстрого вызова **my** (рис. 4.56).

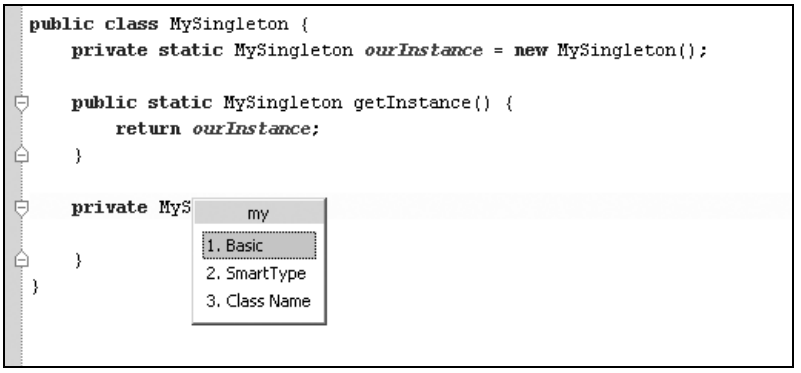


Рис. 4.56. Всплывающее меню с командами из списка быстрого вызова

4.1.16. Диалог настройки *External Tools (Внешние программы)*

В давние времена, когда в IntelliJ IDEA еще не было поддержки плагинов (*см. главу 15*), в качестве средства расширения возможностей программы использовались внешние утилиты (External Tools) — запускаемые программы, вызов которых в IntelliJ IDEA был настроен с помощью диалога, представленного на рис. 4.57.

Несмотря на то, что IntelliJ IDEA давно поддерживает плагины, возможность задать внешние утилиты не устарела. Внешние утилиты удобно использовать, если требуется часто запускать какие-либо программы, особенно

если им требуется передавать в качестве параметров файлы из проекта в IntelliJ IDEA.

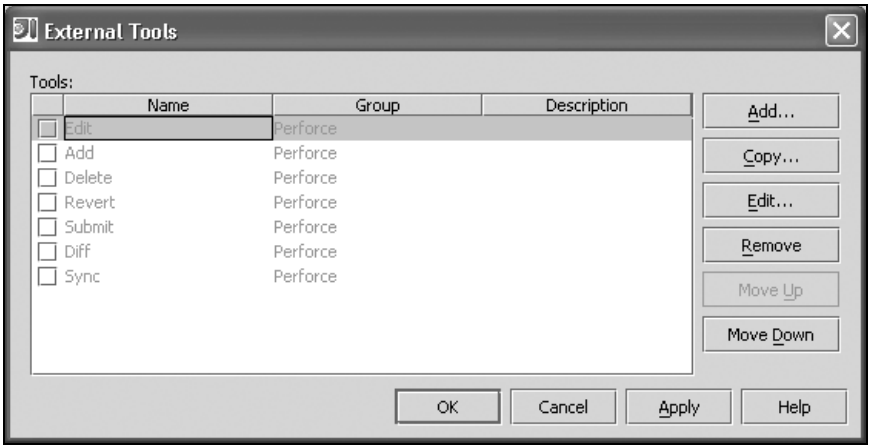


Рис. 4.57. Диалог настройки внешних утилит

Большую часть диалога настройки внешних утилит занимает таблица, в которой представлена следующая информация о внешних утилитах:

- опция выключения утилиты;
- **Name** (Имя) — имя утилиты;
- **Group** (Группа) — группа, к которой относится утилита. Количество утилит не ограничено, и возможность группировать их может оказаться весьма полезной;
- **Description** (Описание) — краткое описание утилиты.

В IntelliJ IDEA имеются настройки для семи внешних утилит, однако все они по умолчанию выключены.

В правой части диалога содержатся кнопки, с помощью которых можно создавать (**Add**), копировать (**Copy**), редактировать (**Edit**), удалять (**Remove**) утилиты, а также менять их расположение в таблице просмотра с помощью кнопок **Move Up** (Вверх) и **Move Down** (Вниз).

4.1.16.1. Создание и редактирование внешней утилиты

Создание и редактирование внешней утилиты производится с помощью диалога, представленного на рис. 4.58.

Для простоты понимания, рассмотрим создание новой внешней утилиты, которая будет возвращать список файлов из указанной директории проекта без учета вложенных директорий.

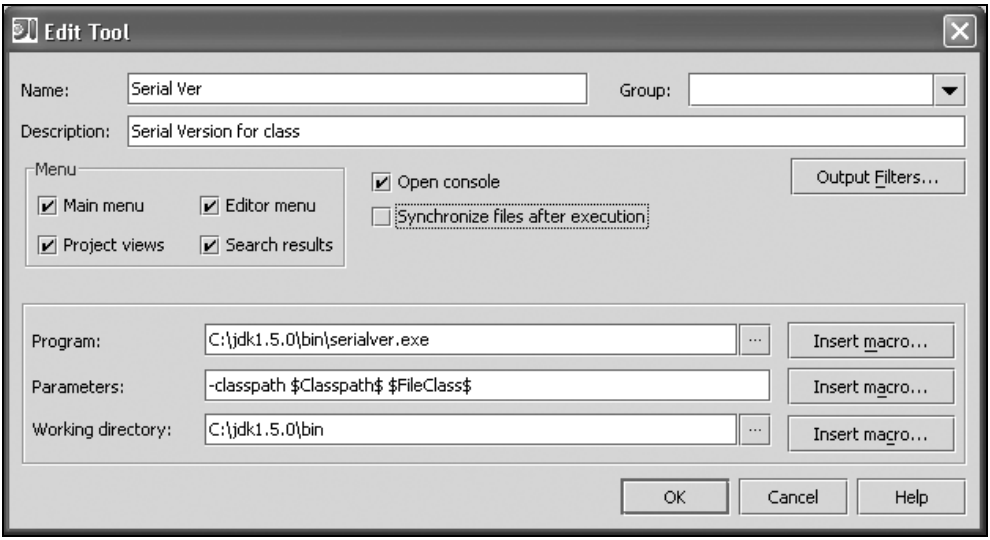


Рис. 4.58. Редактирование внешней утилиты

В диалоге редактирования утилиты можно настроить следующие параметры:

- **Name** (Имя) — имя утилиты;
- **Group** (Группа) — группа утилиты. В качестве группы можно выбрать одну из существующих групп, задать новую, либо не задавать вовсе;
- **Description** (Описание) — краткое описание утилиты;
- **Menu** (Меню) — группа опций, задающих, в меню каких разделов будет добавлена команда вызова утилиты. Если для утилиты задана группа, команда вызова утилиты будет размещена внутри выпадающего подменю. Всего есть четыре опции:
  - **Main menu** (Главное меню) — добавление команды в пункт главного меню **Tools** (Утилиты);
  - **Project views** (Окна просмотра информации о проекте) — команда вызова утилиты будет также добавлена в контекстное меню информации о проекте, представленной в инструментальном окне **Project** (Проект) (*см. раздел 3.4.8.2*);
  - **Editor menu** (Меню редактора) — команда будет добавлена в контекстное меню редактора;
  - **Search results** (Результаты поиска) — команда будет добавлена в контекстное меню с результатами поиска;
- **Open console** (Открывать консоль) — возможность открывать консоль при запуске утилиты. Это бывает полезно, если нам нужна информация, которую утилита будет печатать в консоли;

- **Synchronize files after execution** (Синхронизировать файлы после выполнения) — если утилита изменяет какие-либо файлы, включив эту опцию, можно попросить IntelliJ IDEA синхронизировать файлы проекта после запуска утилиты;
- **Program** (Программа) — строка вызова утилиты;
- **Parameters** (Параметры) — параметры вызова утилиты;
- **Working directory** (Рабочая директория) — директория, в которой будет запущена утилита;
- **Output Filters** (Фильтры вывода) — кнопка, которая позволяет настроить фильтры вывода утилиты. Об этом немного позже.

Макрокоманды

Если вы обратили внимание, на рис. 4.58 рядом с полями ввода информации о программе, ее параметрах и рабочей директории расположены кнопки **Insert macro** (Вставить макрос).

Это не те макросы (макрокоманды), что мы рассматривали в *разделе 3.2.11.7*. Чтобы различать их, мы будем говорить о макросах из данного раздела, как о макросах, а о макрокомандах из *раздела 3.2.11.7*, как о макрокомандах.

Макросы представляют собой именованные обозначения различных параметров проекта, его директорий и файлов.

Нажав на кнопку **Insert macro** (Вставить макрос), можно выбрать макрос из перечисленных (рис. 4.59).

В верхней части диалога содержится список макросов (**Macros**), в нижней — их значение в текущем контексте (**Macro preview**). Значение некоторых макросов не меняется в зависимости от контекста, например, макрос **ProjectName** всегда равен имени проекта.

Далее мы приведен список макросов с кратким описанием для каждого:

- **Classpath** — пути к классам проекта;
- **ClasspathEntry** — составляющая пути классов проекта, к которой принадлежит текущий файл;
- **ColumnNumber** — номер позиции в строке;
- **FileClass** — имя класса файла;
- **FileDir** — директория файла;
- **FileDirRelativeToProjectRoot** — директория файла относительно корневой директории проекта, которому принадлежит файл;
- **FileDirRelativeToSourcepath** — директория файла относительно директории с исходными текстами, в которой находится файл;

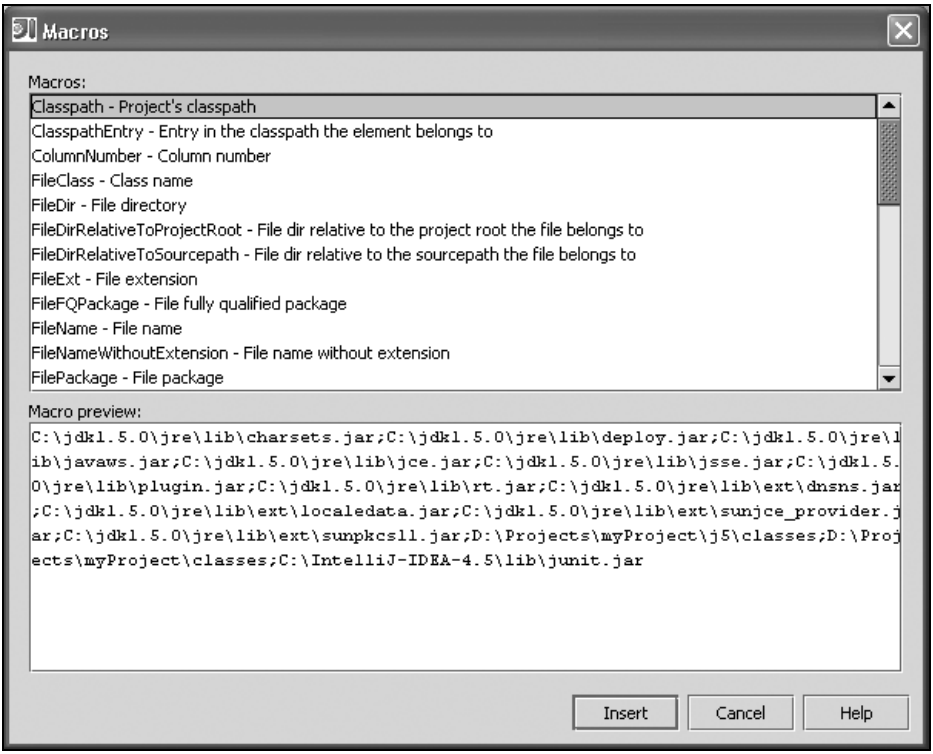


Рис. 4.59. Список макросов

- **FileExt** — расширение файла;
- **FileFQPackage** — полное имя пакета, в котором находится файл;
- **FileName** — имя файла с расширением;
- **FileNameWithoutExtension** — имя файла без расширения;
- **FilePackage** — пакет файла (без префикса пути);
- **FilePath** — полный путь к файлу;
- **FilePathRelativeToProjectRoot** — путь к файлу относительно корневой директории проекта, которому он принадлежит;
- **FilePathRelativeToSourcepath** — путь к файлу относительно директории с исходными текстами, которой принадлежит файл;
- **FileRelativeDir** — директория файла относительно файла проекта;
- **FileRelativePath** — путь к файлу относительно файла проекта;
- **JavaDocPath** — директория вывода JavaDoc (*см. раздел 3.2.11.5*);
- **JDKPath** — корневой каталог JDK (JAVA_HOME);

- **LineNumber** — номер строки;
- **OutputPath** — путь, в котором создаются скомпилированные Java-классы. Если путей несколько, они будут перечислены с использованием системного разделителя пути (в Windows — это ;, в Unix — :);
- **ProjectFileDir** — директория проекта;
- **ProjectFilePath** — путь к файлу проекта;
- **ProjectName** — имя проекта;
- **Projectpath** — пути с исходными текстами проекта;
- **Prompt** — макрос, указывающий IntelliJ IDEA показать диалог с возможностью ввода значения;
- **Sourcepath** — пути с исходными текстами проекта;
- **SourcepathEntry** — одна из записей пути к исходным текстам, к этой записи относится текущий файл;
- **/FileDirRelativeToProjectRoot** — директория файла относительно корневой директории проекта, которому он принадлежит. В качестве разделителей пути используется прямая косая черта /;
- **/FileDirRelativeToSourcepath** — директория файла относительно директории с исходными текстами, которой принадлежит файл. В качестве разделителей пути используется прямая косая черта /;
- **/FilePathRelativeToProjectRoot** — путь к файлу относительно корневой директории проекта, которому он принадлежит. В качестве разделителей пути используется прямая косая черта /;
- **/FilePathrelativeToSourcepath** — путь к файлу относительно директории с исходными текстами, которой принадлежит файл. В качестве разделителей пути используется прямая косая черта /;
- **/FileRelativeDir** — директория файла относительно файла проекта. В качестве разделителей пути используется прямая косая черта /;
- **/FileRelativePath** — путь к файлу относительно файла проекта. В качестве разделителей пути используется прямая косая черта /.

Фильтры вывода

Предназначены для фильтрации информации, выводимой утилитой в консоль, а также для подсветки элементов, соответствующих регулярному выражению, настроенному для фильтра. Фильтры настраиваются с помощью диалога, представленного на рис. 4.60.

Фильтры можно добавлять (кнопка **Add**), редактировать (кнопка **Edit**), удалять (кнопка **Remove**), а также изменять положение фильтров в списке.

Создание и редактирование фильтра производится с помощью диалога, представленного на рис. 4.61.

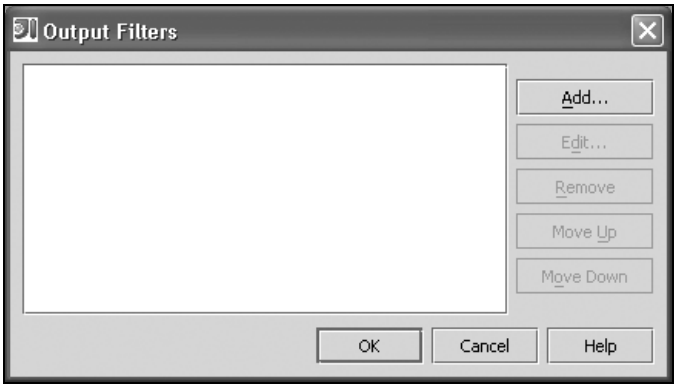


Рис. 4.60. Диалог настройка фильтров вывода

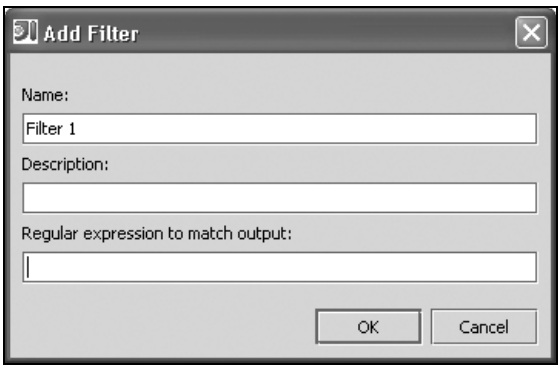


Рис. 4.61. Диалог редактирования фильтра вывода

Примечание

К сожалению, нам так и не удалось придумать хорошего примера использования фильтров.

4.1.16.2. Запуск внешней утилиты

На рис. 4.58 представлен диалог создания утилиты для генерации строки `serialVersionUID` для сериализуемого класса с помощью утилиты serialver из стандартной поставки JDK. Утилите передаются два параметра:

- `-classpath $Classpath$`;
- `$FileClass$`.

Первый параметр задает пути к классам, в которых утилита будет искать класс, а второй параметр — это имя самого класса.

После создания данная утилита будет доступна для вызова из пункта **Tools** (Утилиты) главного меню, а также из контекстных меню.

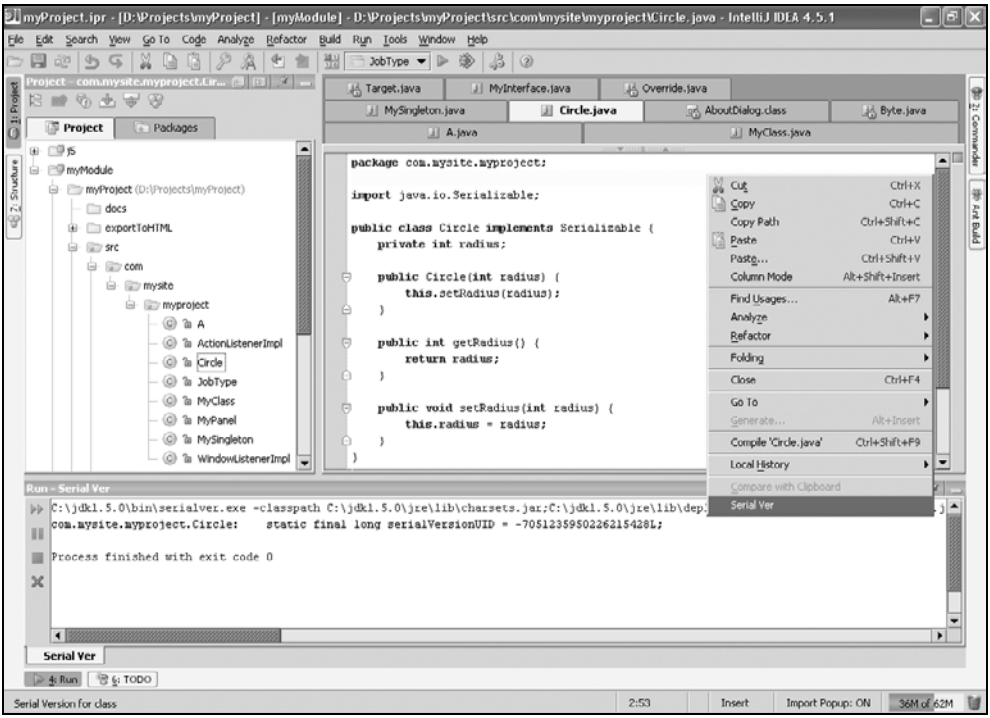


Рис. 4.62. Вызов внешней утилиты и результат ее работы

Чтобы запустить созданную нами утилиту Serial Ver, нужно либо в редакторе, либо в одном из соответствующих инструментальных окон открыть класс, имплементирующий интерфейс Serializable, для него можно будет вызвать из контекстного меню команду **Serial Ver**, соответствующую нашей утилите (рис. 4.62).

Вывод утилиты направлен в консоль, открытую в инструментальном окне **Run** (Запуск), в нижней части окна IntelliJ IDEA.

Сгенерированная строка с serialVersionUID напечатана в консоли, осталось только скопировать ее в тело класса.

4.1.17. Диалог настройки *Path Variables (Местоположения)*

Диалог настройки местоположения (или переменных путей) представлен на рис. 4.63.

Местоположения — это именованные значения путей на диске, которые могут быть использованы для конфигурирования проекта.

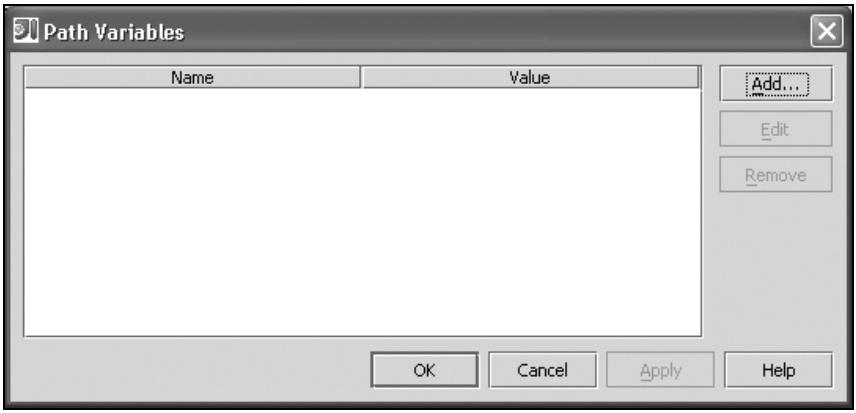


Рис. 4.63. Диалог настройки переменных путей

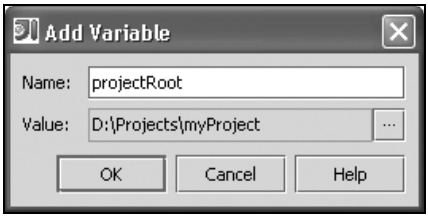


Рис. 4.64. Добавление переменной пути

У переменной пути есть имя и значение. Чтобы создать новую переменную, нужно нажать кнопку **Add** (Добавить) и в открывшемся диалоге задать название переменной и ее значение (рис. 4.64).

Переменные пути также можно редактировать, меняя их имена и значения (для этого нужно нажать кнопку **Edit** (Редактировать)), а также удалять с помощью кнопки **Delete** (Удалить).

Переменные пути удобно использовать при разработке одного проекта несколькими разработчиками, когда у каждого разработчика локальные пути к файлам проекта или библиотекам отличаются. В этом случае пути в проекте настраиваются относительно переменных путей, которые у каждого разработчика настроены персонально.

4.1.18. Диалог *TODO (Заметки)*

Многие программисты пишут в коде программы комментарии с напоминаниями или указаниями себе или другим программистам на то, что нужно сделать вроде тех, что представлены в листинге 4.16.

Листинг 4.16

```
/*
    ToDo: Chnage class implementation to keep float radius
    CriticalToDo: implement methods readObject() and writeObject()
    [John Smith]: please, implement it
*/
```

Используя образцы (patterns), которые настраиваются с помощью диалога, представленного на рис. 4.65, можно указать IntelliJ IDEA выделять подобные комментарии специальном образом (рис. 4.66).

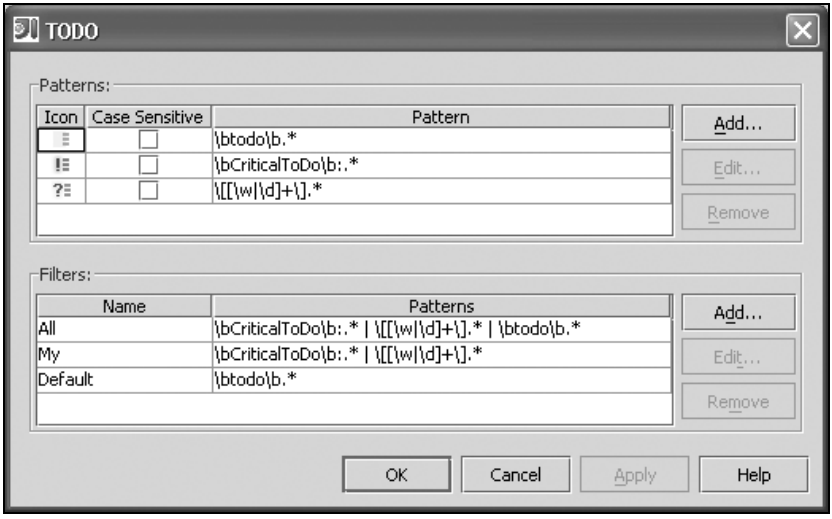


Рис. 4.65. Настройка образцов для списка дел

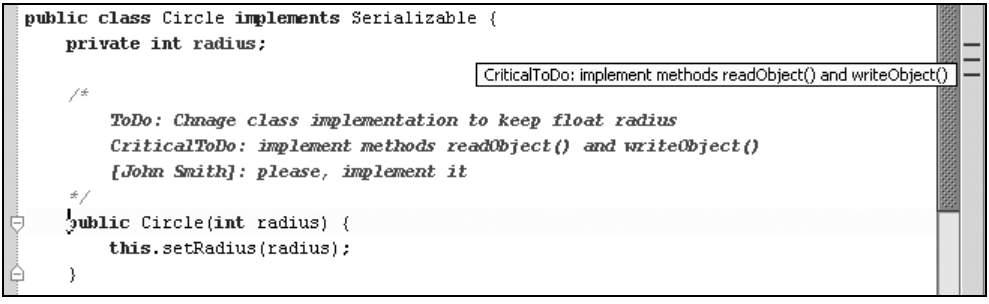


Рис. 4.66. Комментарии в коде, выделенные специальным образом на основе образцов

Кроме выделения в тексте комментария, IntelliJ IDEA выводит на полях редактора метки синего цвета, которые указывают на такие комментарии со списком дел.

Диалог настройки образцов состоит из двух панелей:

- **Patterns** (Образцы) — таблица образцов. В таблице представлена следующая информация об образцах:
  - **Icon** (Пиктограмма) — пиктограмма, соответствующая данному образцу. Эта пиктограмма отображается при просмотре списка дел в инструментальном окне **TODO** (Заметки);
  - **Case Sensitive** (Чувствительность к регистру) — опция, включив которую можно задать режим работы образцов с соблюдением или без соблюдения регистра символов;
  - **Pattern** (Образец) — регулярное выражение, на основании которого работает поиск комментариев со списком дел;
- **Filters** (Фильтры) — таблица-список фильтров, созданных на основе образцов. В таблице представлена следующая информация:
  - **Name** (Название) — название фильтра;
  - **Patterns** (Образец) — список образцов, включенных в фильтр.

Также в правой части диалога расположены кнопки для добавления (**Add**), редактирования (**Edit**), удаления (**Remove**) образцов и фильтров.

4.1.18.1. Создание и редактирование образцов

Для создания и редактирования образцов используется диалог, представленный на рис. 4.67.

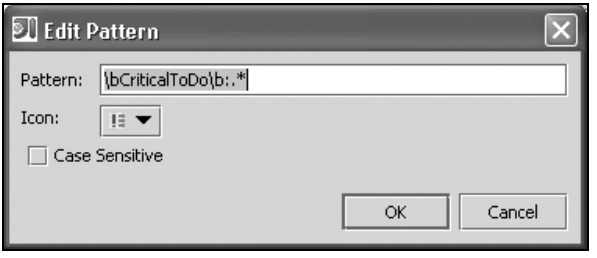


Рис. 4.67. Редактирование образца

Для образца можно задать три параметра:

- **Pattern** (Образец) — регулярное выражение, на основе которого будет осуществляться поиск комментариев со списком дел. Синтаксис регулярного выражения соответствует правилам регулярных выражений для `java.util.regex`;

- **Icon** (Пиктограмма) — пиктограмма, с которой комментарии будут представлены в инструментальном окне **TODO** (Заметки);
- **Case Sensitive** (Чувствительность к регистру) — опция, включающая чувствительность к регистру при поиске комментариев.

4.1.18.2. Фильтры

Фильтры образцов нужны исключительно в целях повышения производительности. Они позволяют скрывать или отображать пункты из списка дел в зависимости от того, какие образцы входят в тот или иной фильтр.

На рис. 4.65 в списке фильтров имеется три фильтра:

- **All** (Все) — включает в себя все образцы;
- **My** (Мои) — некоторый пользовательский фильтр, который включает только два образца из приведенных;
- **Default** (По умолчанию) — фильтр, использующий только образец, заданный в IntelliJ IDEA по умолчанию.

Между этими фильтрами можно переключаться (*см. раздел 4.1.18.5*).

4.1.18.3. Примеры образцов

В IntelliJ IDEA имеется один предустановленный образец с регулярным выражением `\btodo\b.*` — все комментарии, содержащие отдельно написанное слово '`todo`', а также вся строка за ним. Регистр символов не учитывается.

В листинге 4.13 этим образцам выделена первая строка комментария.

Нами были добавлены два дополнительных образца с регулярными выражениями:

- `\bCriticalToDo\b:.*` — аналогично встроенному образцу, но реагирует на слово '`CriticalToDo`', написанное в любом регистре. Образцом с этим регулярным выражением отмечена вторая строка комментария из листинга 4.13;
- `\[[\w\d\s]+\].*` — комментарии вида '[имя фамилия]', которые часто пишут программисты в коде, либо чтобы подписаться под каким-то изменением, либо чтобы указать другому программисту на это место в коде. Образец с этим регулярным выражением воздействовал на последнюю строку комментария.

4.1.18.4. Просмотр списка дел для всего проекта

В IntelliJ IDEA имеется возможность просмотреть список дел (выделенных образцами) для всего проекта. Для этого используется специальное инструментальное окно **TODO** (Заметки) (рис. 4.68).

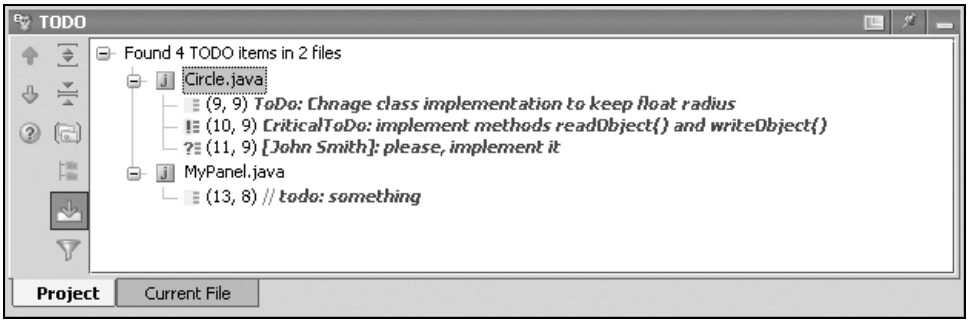


Рис. 4.68. Просмотр списка дел для всего проекта

В проектах, где не используются системы учета требований (Change Request — требование к изменению) для разработки, с помощью списка дел можно реализовать обмен такими требованиями между разработчиками.

В инструментальном окне **TODO** (Заметки) есть возможность просмотреть список дел как для всего проекта (закладка **Project**), так и для конкретного файла (закладка **Current File**). Пункты из списка выводятся вместе с пиктограммами, которые были настроены для образцов, нашедших соответствующие пункты дел.

4.1.18.5. Переключение фильтров

На панели инструментов окна **TODO** (Заметки) имеется кнопка **Filter TODO Items** (Фильтровать заметки) (см. рис. 4.68). Если на нее нажать, будет отображен список доступных фильтров, между которыми можно будет переключиться (рис. 4.69). Таким образом, в список дел попадут не все комментарии, а лишь те, что настроены для фильтра.

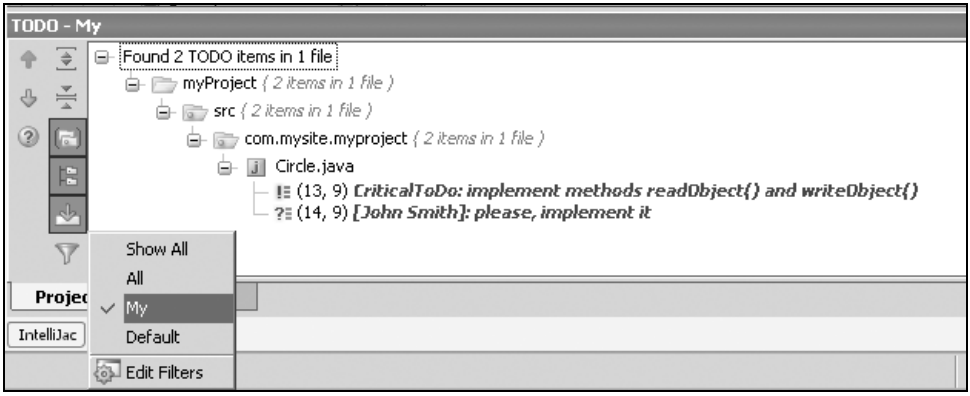


Рис. 4.69. Выбор фильтра для образцов

4.1.19. Диалог настройки *Plugins (Плагины)*

Для удобства загрузки плагинов и их обновления, а также удаления их из IntelliJ IDEA предусмотрен менеджер плагинов, представленный на рис. 4.70.

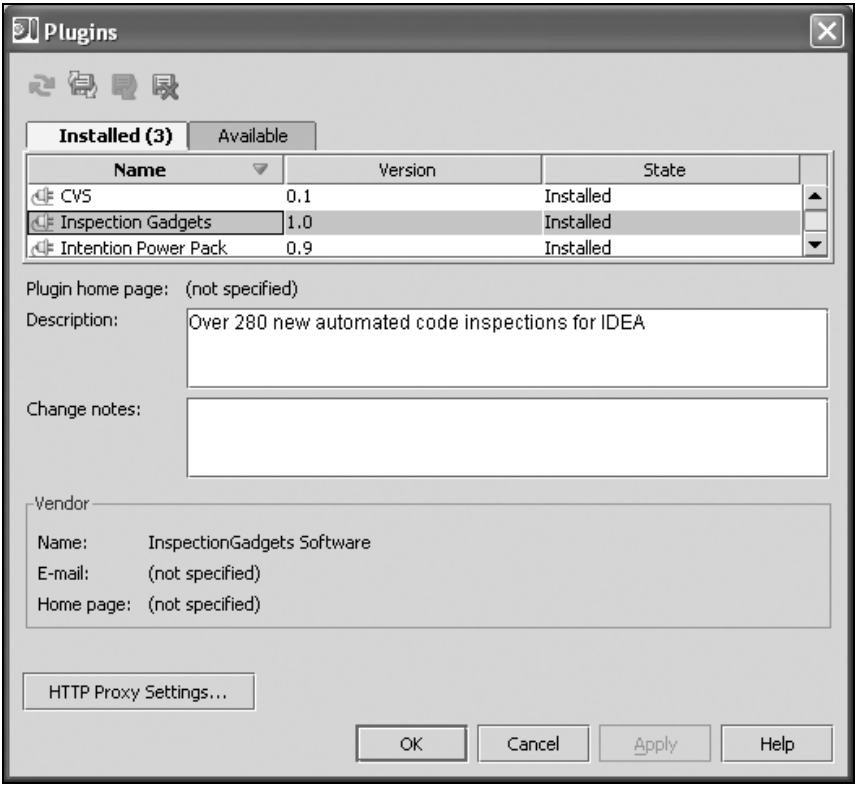


Рис. 4.70. Менеджер плагинов

Диалог менеджера плагинов содержит две закладки:

- **Installed** (Установленные плагины);
- **Available** (Доступные плагины).

Для просмотра информации о плагине его необходимо выделить и в нижней области окна появится информация о плагине — URL страницы проекта, описание, история изменений, информация об авторе или фирме.

Закладка **Available** (Доступные) показывает, какие плагины можно установить. Список доступных плагинов скачивается с сервера компании JetBrains, поэтому для просмотра доступных плагинов необходимо подключение к Интернету.

Если в вашей корпоративной сети применены политики доступа к Web-ресурсам, вам понадобится настроить использование вашего корпоративно-

го прокси-сервера. Для этого в главном окне менеджера плагинов нажмите кнопку **HTTP Proxy Settings** (Настройки HTTP-прокси).

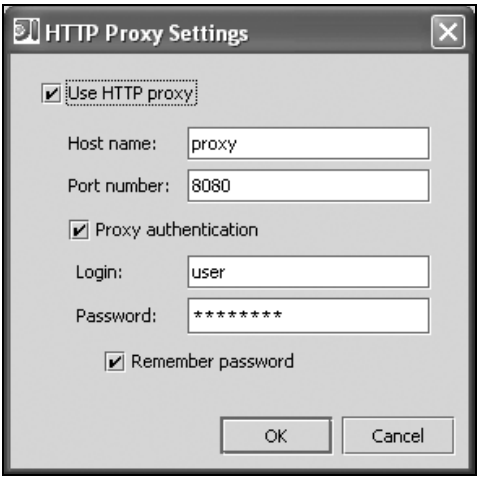


Рис. 4.71. Настройка HTTP-прокси

В диалоге настройки прокси (рис. 4.71) имеется возможность ввести следующие параметры:

- **Use HTTP proxy** (Использовать HTTP-прокси) — опция позволяет включить использование прокси;
- **Host name** (Имя хоста) — имя хоста прокси;
- **Port number** (Номер порта) — номер порта прокси;
- **Proxy authentication** (Аутентификация на сервере прокси) — опция включения аутентификации для доступа к прокси;
- **Login** (Логин);
- **Password** (Пароль);
- **Remember password** (Запоминать пароль) — опция позволяет сохранять пароль для доступа к прокси.

После загрузки списка плагинов показывается таблица, в которой представлены все доступные плагины (рис. 4.72).

В таблице представлен также рейтинговый показатель — число загрузок плагина (**Downloads**) с сервера плагинов. Достаточно отсортировать в обратном направлении по этому столбцу, чтобы знать, какие плагины наиболее популярны.

Используя менеджер плагинов, можно добавлять новые, обновлять и удалять существующие плагины. Все действия над плагинами осуществляются с помощью панели инструментов, а также посредством контекстного меню.

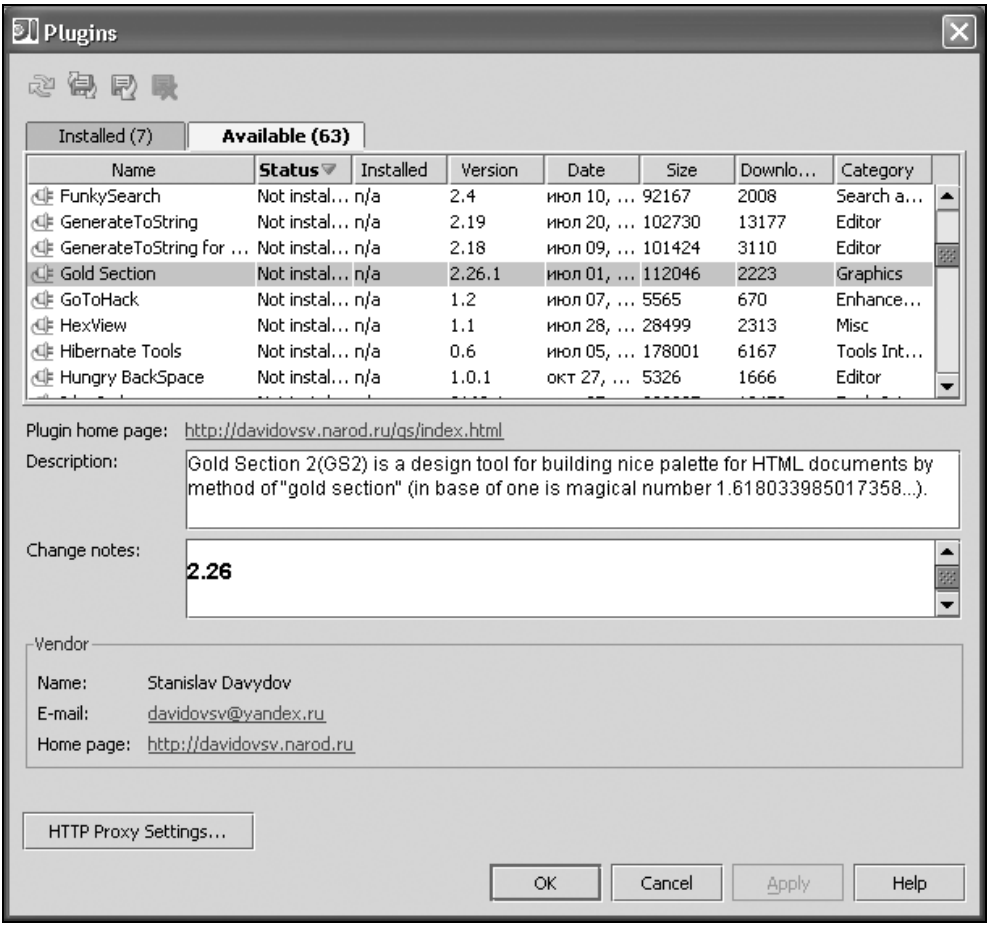


Рис. 4.72. Список доступных плагинов

На панели инструментов представлены следующие команды:

- **Synchronize with Plugin Repository** (Получить обновленный список плагинов с сервера) — <Ctrl>+<Alt>+<Y>;
- **Update Installed Plugins** (Обновить установленные плагины) — команда позволяет проверить наличие новых версий установленных плагинов, а если таковые будут найдены, то и скачать их;
- **Download and Install Plugin** (Скачать и установить плагин);
- **Uninstall Plugin** (Удалить плагин из IntelliJ IDEA).

В IntelliJ IDEA имеются семь предустановленных плагинов:

- **CVS** — интеграция с системой контроля версий CVS (*см. раздел 12.3*);
- **Inspection Gadgets** — набор инспекций кода (*см. раздел 6.4.1*);

- **Intension Power Pack** — набор "подсказок" (*см. разделы 6.1.3, 4.1.21*);
- **JSR 45 Integration** — интеграция JSR-45 совместимого сервера приложений (*см. раздел 9.2.6*);
- **Starteam** — интеграция с системой контроля версий Starteam (*см. раздел 12.5*);
- **Tomcat** — интеграция с сервером приложений Tomcat (*см. раздел 9.2.5*);
- **Weblogic** — интеграция с сервером приложений WebLogic (*см. раздел 9.2.7*).

Дополнительные возможности плагинов IntelliJ IDEA подробно рассмотрены в *главе 15*.

4.1.20. Диалог настройки *Application Servers (Серверы приложений)*

Диалог настройки серверов приложений представлен на рис. 4.73.

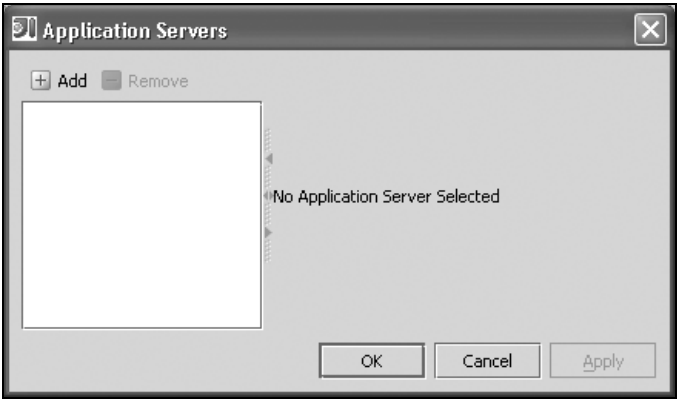


Рис. 4.73. Диалог настройки серверов приложений

В левой части диалога содержится список доступных (настроенных) серверов приложений, а в правой — их свойства.

Добавить сервер приложений можно с помощью кнопки со знаком +. В ответ на ее нажатие открывается выпадающее меню со списком из трех доступных типов серверов приложений (рис. 4.74):

- **Tomcat Server** — Tomcat Servlet/JSP контейнер — сервер проекта Apache Jakarta;
- **WebLogic Server** — сервер приложений компании Bea;
- **Generic** — некоторый абстрактный сервер приложений, задаваемый только набором библиотек.

Рассмотрим настройку серверов этих типов.

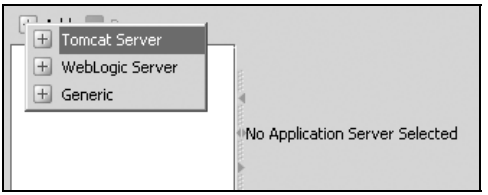


Рис. 4.74. Варианты создания сервера приложений

4.1.20.1. Сервер Tomcat

После выбора из списка сервера Tomcat отображен диалог, в котором IntelliJ IDEA предложит настроить свойства Tomcat (рис. 4.75).

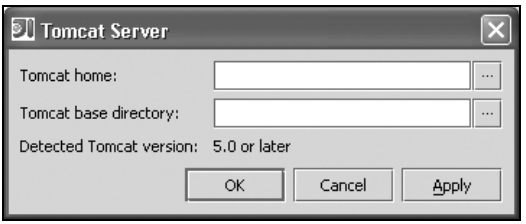


Рис. 4.75. Настройка параметров сервера Tomcat

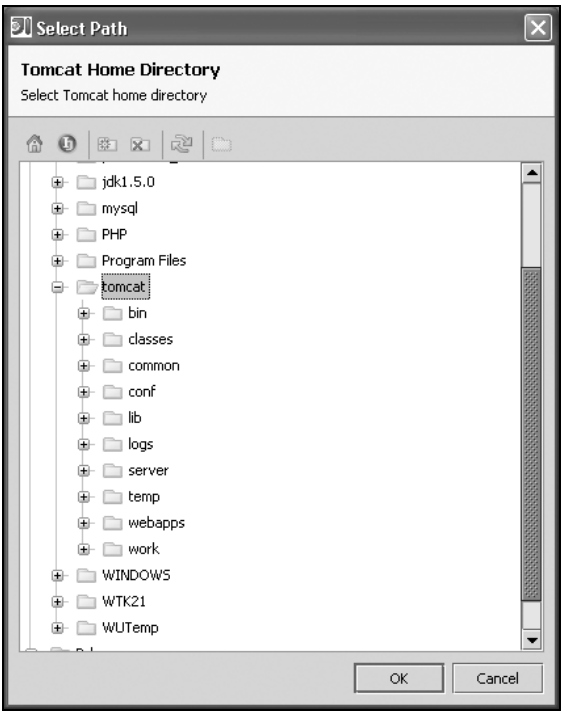


Рис. 4.76. Выбор директории с установленным сервером Tomcat

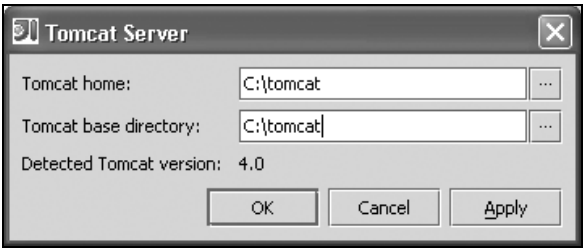


Рис. 4.77. Параметры сервера Tomcat настроены

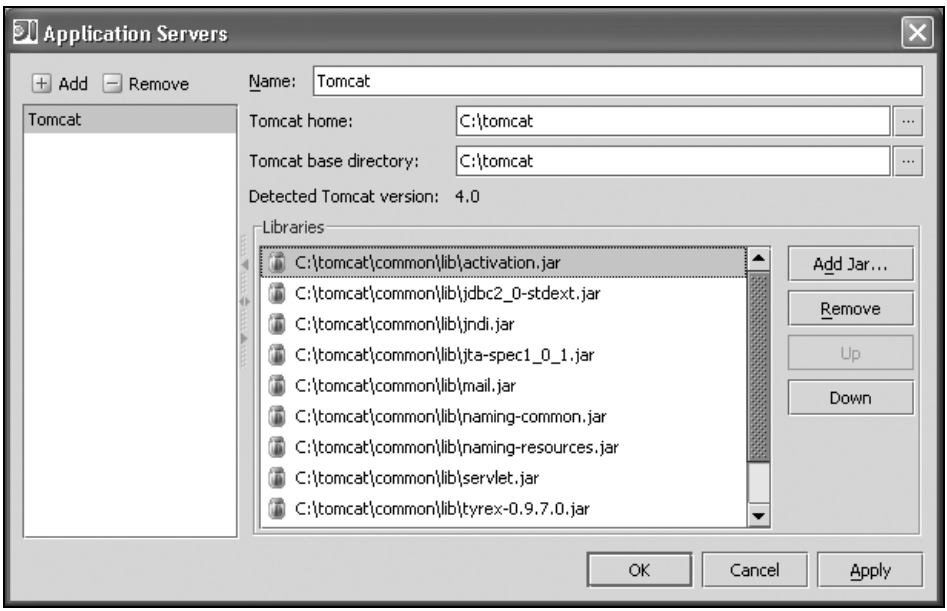


Рис. 4.78. Информация о сервере Tomcat

Эти свойства не обязательно прописывать вручную. Можно выбрать директорию с установленным сервером Tomcat (**Tomcat home**), нажав на кнопку с... (с многоточием) (рис. 4.76).

Как только директория выбрана, IntelliJ IDEA автоматически настроит и второй параметр **Tomcat base directory** (Исходная директория Tomcat) (рис. 4.77).

После настройки параметров сервера IntelliJ IDEA автоматически отобразит версию сервера Tomcat в поле **Detected Tomcat version** (Обнаруженная версия Tomcat).

После подтверждения диалога информация о сервере Tomcat будет добавлена в общий список серверов приложений (рис. 4.78).

Для сервера Tomcat представлен список библиотек, которые необходимы для его работы. В большинстве случаев настраивать этот список вручную не требуется.

4.1.20.2. Сервер WebLogic

Если у вас уже установлен сервер WebLogic версии 6.x или выше, IntelliJ IDEA сама попробует определить, какая версия у вас установлена (она ищет специальный файл registry.xml, расположенный в корневой директории установки сервера WebLogic).

Для добавления конфигурации сервера необходимо указать каталог, в котором установлен WebLogic (рис. 4.79).

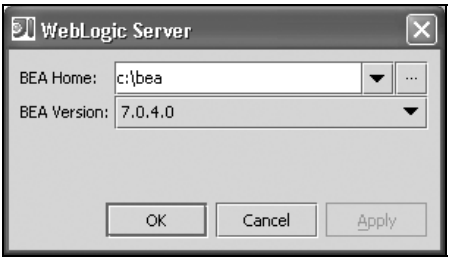


Рис. 4.79. Добавление конфигурации сервера WebLogic

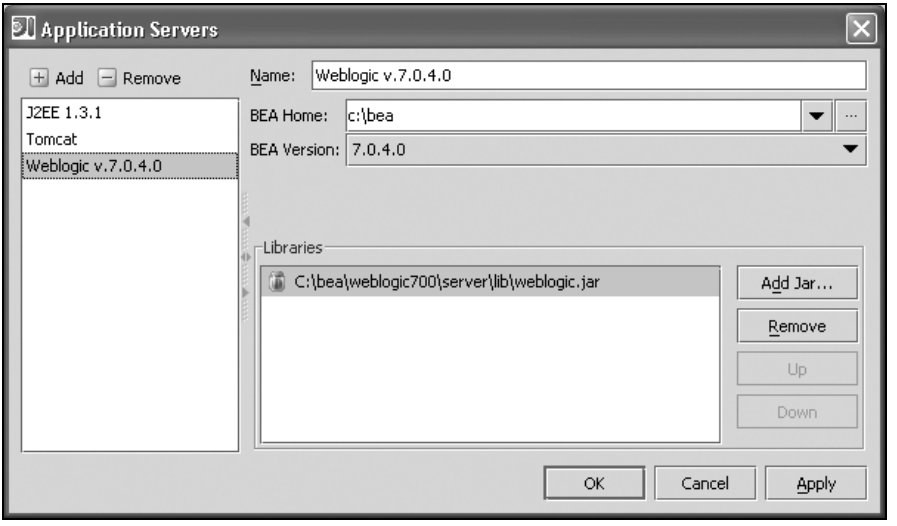


Рис. 4.80. Сервер WebLogic 7.0.4.0 добавлен

Если в указанной директории у вас установлено несколько версий сервера, с помощью поля **BEA Version** можно указать ту, с которой вы хотите работать.

После подтверждения диалога указанный сервер будет добавлен (рис. 4.80).

4.1.20.3. Сервер Generic

В качестве примера абстрактного сервера приложений мы решили взять стандартную поставку J2EE 1.3.1 фирмы Sun.

Выбрав в меню добавления сервера пункт **Generic**, мы добавим абстрактный сервер, для которого потребуется указать набор библиотек (рис. 4.81).

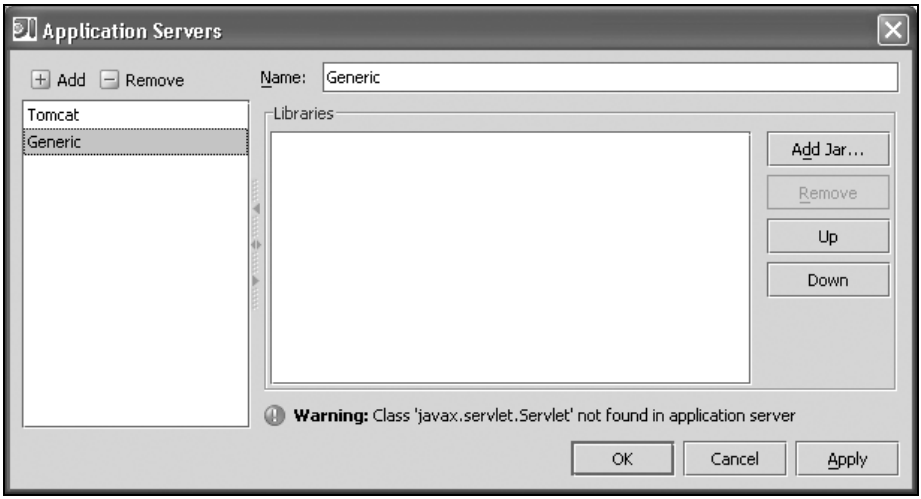


Рис. 4.81. Добавления сервера Generic

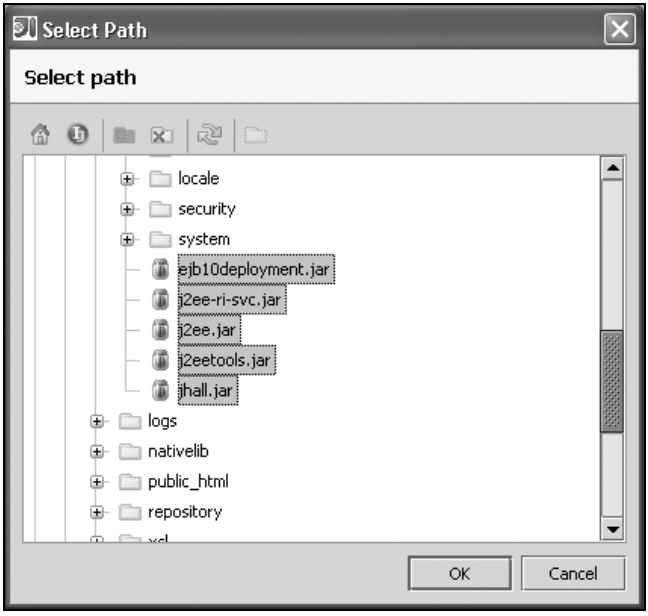


Рис. 4.82. Добавление библиотек из поставки J2EE 1.3.1

В нижней части окна IntelliJ IDEA выводит предупреждающие подсказки, если каких-то библиотек не хватает. В данном случае IntelliJ IDEA сообщает об отсутствии библиотеки с классом `javax.servlet.Servlet`.

Добавим библиотеки из поставки J2EE 1.3.1 (рис. 4.82).

И вот, наш абстрактный сервер приложений готов к использованию (рис. 4.83) — мы можем создавать J2EE-модули с его участием.

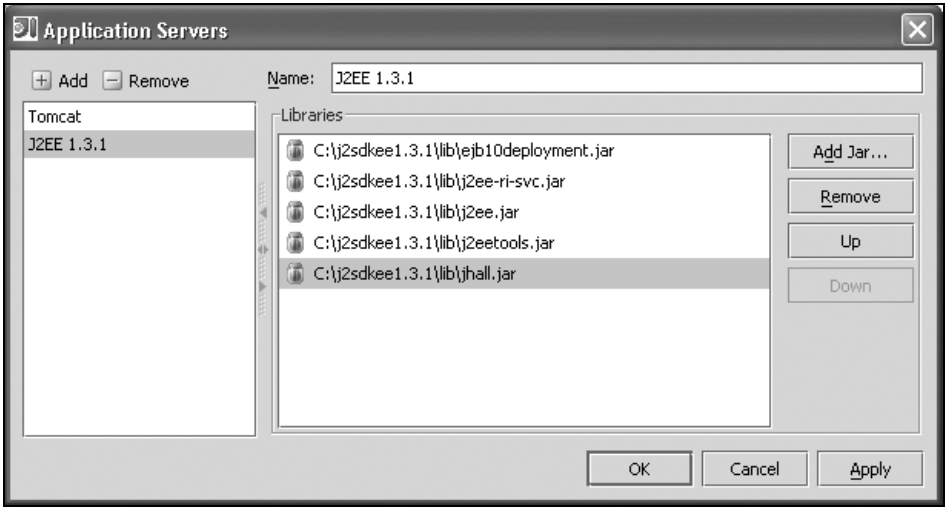


Рис. 4.83. Абстрактный сервер приложений добавлен

4.1.21. Диалог настроек *Intention Power Pack (Дополнительный пакет "подсказок")*

Последний пункт настройки **Intention Power Pack** (Дополнительный пакет "подсказок") предназначен для настройки дополнительных "подсказок" (рис. 4.84). Подробнее об использовании "подсказок" можно прочитать в *разделе 6.1.3*.

В диалоге представлен набор опций:

- **Enable convert multiply/divide <-> shift intentions** — разрешить "подсказки" прямой и обратной конвертации выражений с умножением/делением в выражения со сдвигом;
- **Enable convert '&&' <-> '||' intentions** — разрешить "подсказки" прямой и обратной конвертации логических операторов `'&&'` и `'||'`;
- **Enable flip '&&' and flip '||' intentions** — разрешить "подсказки" взаимного обмена логических операторов `'&&'` и `'||'`;

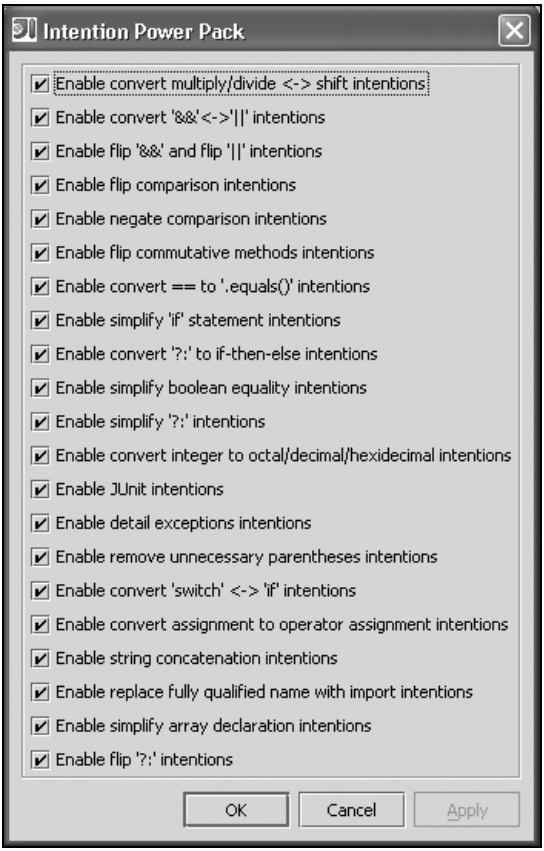


Рис. 4.84. Настройки дополнительного пакета "действий намерения"

- **Enable flip comparison intentions** — разрешить "подсказки" взаимного обмена между операторами сравнения;
- **Enable negate comparison intentions** — разрешить "подсказки" сравнения с отрицанием;
- **Enable flip commutative methods intentions** — разрешить "подсказки" обмена коммутативных методов;
- **Enable convert '==' to '.equals()' intentions** — разрешить "подсказки" конвертации оператора сравнения '==' в вызов метода '`.equals()`';
- **Enable simplify 'if' statement intentions** — разрешить "подсказки" упростить оператор `if`;
- **Enable convert '?:' to if-then-else intentions** — разрешить "подсказки" конвертировать тернарный оператор `?:` в оператор сравнения `if-then-else`;
- **Enable simplify boolean equality intentions** — разрешить "подсказки" упрощать булевские равенства;

- **Enable simplify '?:' intentions** — разрешить "подсказки" упрощать тернарный оператор `?:`;
- **Enable convert integer to octal/decimal/hexidecimal intentions** — разрешить "подсказки" конвертации представления целых чисел между восьмеричным, десятичным и шестнадцатеричным;
- **Enable JUnit intentions** — разрешить "подсказки" для тестов JUnit;
- **Enable detail exceptions intentions** — разрешить "подсказки" детализирования исключений. Например, если имеется код, представленный в листинге 4.17, данная группа "подсказок" предложит привести этот код к виду, представленному в листинге 4.18;

Листинг 4.17

```
try {
    FileOutputStream fos = new FileOutputStream("test.txt");
    String str = new String(new byte[] {}, "cp1251");
} catch (Exception e) {
    //
}
```

Листинг 4.18

```
try {
    FileOutputStream fos = new FileOutputStream("test.txt");
    String str = new String(new byte[] {}, "cp1251");
} catch (FileNotFoundException e) {
    //
} catch (UnsupportedEncodingException e) {
    //
}
```

- **Enable remove unnecessary parentheses intentions** — разрешить "подсказки" об удалении лишних круглых скобок;
- **Enable convert 'switch' <-> 'if' intentions** — разрешить "подсказки" взаимной конвертации между операторами `switch` и `if`;
- **Enable convert assignment to operator assignment intentions** — разрешить "подсказки" конвертации оператора присваивания в присваивающий оператор, например, код `i = i+5` может быть сконвертирован в `i+=5`;
- **Enable string concatenation intentions** — разрешить "подсказки" конкатенации строк;
- **Enable replace fully qualified name with import intentions** — разрешить "подсказки" заменять полные имена на строки импорта;

- **Enable simplify array declaration intentions** — разрешить "подсказки" упрощения определения массивов;
- **Enable flip '?:' intentions** — разрешить "подсказки" обмена между частями тернарного оператора `?:`.

Подробнее про "подсказки" можно прочитать в *разделе 6.4.1.*

4.2. Настройки проекта

Настройки проекта представлены пятью кнопками:

- **Paths** (Пути) — настройка модулей проекта;
- **Compiler** (Компилятор) — настройка компилятора;
- **Version Control** (Системы контроля версий) — настройка системы контроля версий для проекта;
- **Project Code Style** (Стилевые настройки кода для проекта) — индивидуальные стилевые настройки кода для проекта;
- **GUI Designer** (Редактор пользовательского интерфейса) — настройка визуального редактора.

Данные настройки имеют место при настройке любого проекта, а также при настройке шаблона проекта (*см. раздел 3.2.1.12*).

4.2.1. Диалог настройки *Paths (Пути)*

Данный диалог предназначен для настройки модулей проекта, которая будет более подробно рассмотрена в *главе 5.* Здесь же мы остановимся лишь на двух опциях данного диалога, имеющих отношение к проекту в целом.

Общий вид диалога настройки путей представлен на рис. 4.85.

Нас в данный момент интересуют две опции, расположенные в нижней части диалога:

- **For files outside project file directory** (Для файлов, расположенных за пределами директории с файлом проекта):
  - **Use absolute path** (Использовать абсолютный путь);
  - **Use relative path** (Использовать относительный путь);
- **Language level for project (effective on restart)** (Версия языка Java для проекта) — изменение данной опции имеет эффект при рестарте проекта:
  - **1.3** (и ниже);
  - **1.4 — 'assert' keyword** (используется ключевое слово `'assert'`);
  - **5.0 — 'enum' keyword, autoboxing, etc.** (новые возможности языка Java 5).

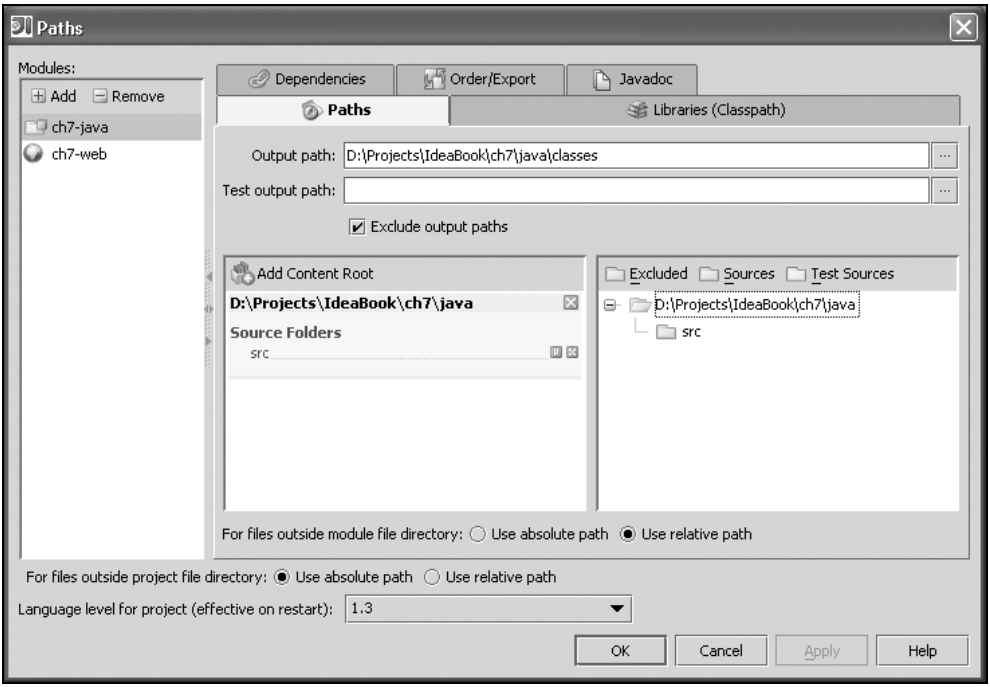


Рис. 4.85. Настройка свойств проекта и модулей

Первую опцию имеет смысл устанавливать в значение относительных путей для проектов, в разработке которых задействовано несколько программистов. Локальные пути на компьютере у каждого из них могут отличаться от других. Эти пути записаны в файле проекта.

Вторую опцию следует установить в значение, соответствующее использованной функциональности языка Java.

Дальнейшие настройки из данного диалога будут рассмотрены в *главе 5*.

4.2.2. Диалог настройки *Compiler (Компилятор)*

Общие настройки компилятора проекта производятся с помощью диалога, представленного на рис. 4.86.

Данный диалог содержит несколько общих опций, а также специальные опции для разных типов компилятора.

Подробнее о процессе компиляции в IntelliJ IDEA можно прочитать в *разделе 9.1*.

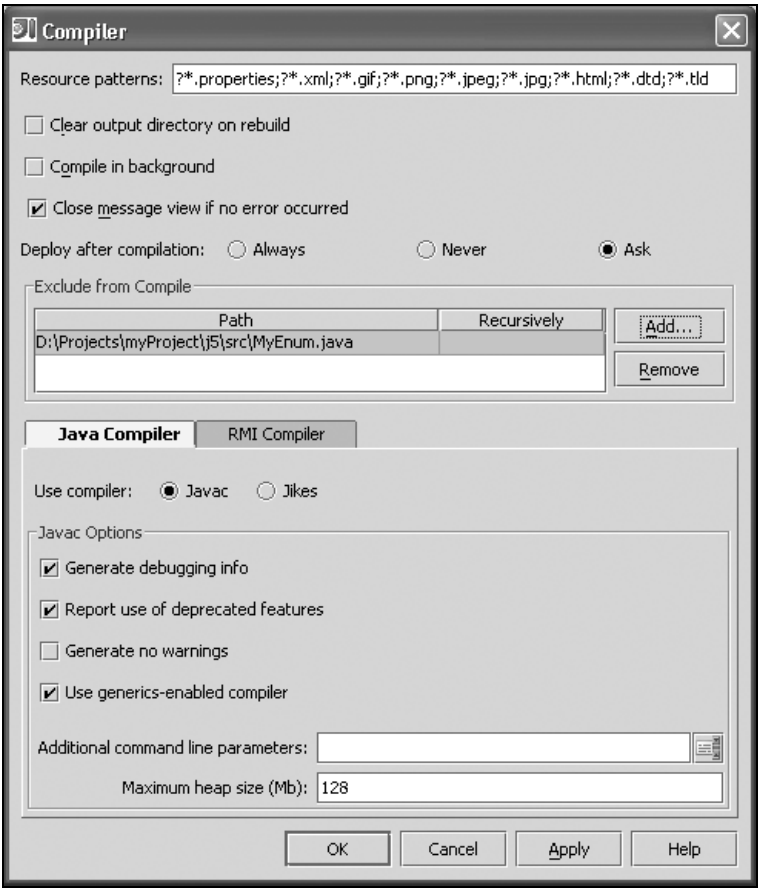


Рис. 4.86. Диалог настройки компилятора проекта

4.2.2.1. Общие настройки компилятора

- **Resource patterns** (Образцы ресурсов) — регулярное выражение в формате `java.util.regex`, или перечисленные через запятую маски файлов, которые IntelliJ IDEA будет распознавать как ресурсы приложения. Эти ресурсы будут скопированы при компиляции в выходной путь в соответствующие пакетам директории. По умолчанию настроены маски файлов со следующими расширениями:
  - properties;
  - xml;
  - gif;
  - png;

- jpeg;
- jpg;
- html;
- dtd;
- tld;

- **Clear output directory on rebuild** (Очищать выходную директорию при перестроении) — данная опция позволяет указать IntelliJ IDEA стирать все файлы, классы и каталоги из выходной директории перед перестроением проекта (*см. разделы 3.2.9.1, 5.4.1*);
- **Compile in background** (Компилировать в фоновом режиме) — опция позволяет всегда запускать компиляцию в фоновом режиме (*см. раздел 5.4.1*);
- **Close message view if no error occurred** (Закрывать просмотр сообщений, если не произошло ошибок компиляции) — опция позволяет автоматически закрывать инструментальное окно **Messages** (Сообщения) (*см. раздел 3.4.8.1*) в случае, если при компиляции не было найдено ошибок или замечаний;
- **Deploy after compilation** (Выкладывать после компиляции) — в случае, если в проекте имеются настроенные Web, EJB или Application Server-модули, для которых требуется отдельная операция выкладывания классов и ресурсов на сервер, данная опция позволяет включить такое выкладывание (deployment) автоматически. Данная опция имеет три возможных значения:
  - **Always** (Всегда) — всегда выкладывать;
  - **Never** (Никогда) — никогда не выкладывать (подразумевает выкладывание в ручном режиме);
  - **Ask** (По запросу) — выкладывать по запросу;
- **Exclude from Compile** (Исключить из компиляции) — в данный список можно включить те файлы, которые не должны быть скомпилированы вместе с проектом. В качестве элементов списка можно задать директории или конкретные файлы. Файлы и директории добавляются в список с помощью кнопки **Add** (Добавить), которая открывает диалог (рис. 4.87). Удалить исключенные файлы и директории из списка можно с помощью кнопки **Remove** (Удалить).

В нижней части диалога настройки компилятора находятся две закладки, на которых расположены опции настройки отдельных типов компиляторов.

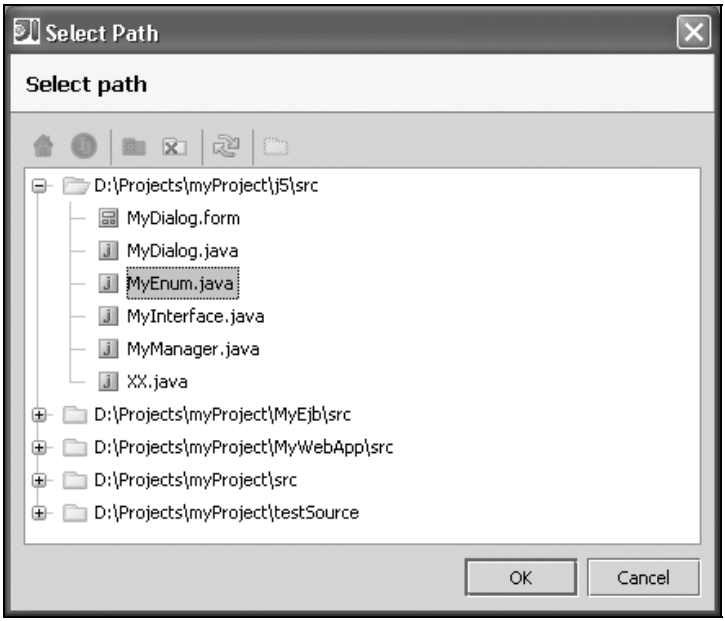


Рис. 4.87. Выбор пути для исключения из компиляции

4.2.2.2. Закладка *Java Compiler (Компилятор Java)*

IntelliJ IDEA поддерживает два компилятора Java:

- **Javac** — компилятор фирмы Sun;
- **Jikes** — компилятор фирмы IBM.

Javac

Компилятор Javac имеет следующие настройки:

- **Generate debugging info** (Генерировать отладочную информацию);
- **Report use of deprecated features** (Сообщать об использовании устаревших (`@deprectaed`) классов и методов);
- **Generate no warnings** (Не сообщать об ошибках);
- **Use generics-enabled compiler** (Использовать компилятор, поддерживающий параметризованные классы) (*см. раздел 3.2.11.6, главу 11*);
- **Additional command line parameters** (Дополнительные параметры командной строки);
- **Maximum heap size** (Максимальный размер памяти) — максимальный размер памяти в мегабайтах, выделяемой компилятору.

Jikes

Компилятор Jikes имеет следующие настройки:

- **Path to Jikes executable** (Путь к исполняемому файлу Jikes);
- **Generate debugging info** (Генерировать отладочную информацию);
- **Generate no warnings** (Не сообщать об ошибках);
- **Report use of deprecated features** (Сообщать об использовании устаревших (`@deprectaed`) классов и методов);
- **Additional command line parameters** (Дополнительные параметры командной строки).

4.2.2.3. Закладка *RMI Compiler (Компилятор RMI)*

Компилятор RMI имеет следующие опции:

- **Enable RMI stubs generation** (Разрешить генерировать заготовки RMI);
- **Generate IIOP stubs** (Генерировать заготовки IIOP);
- **Generate debugging info** (Генерировать отладочную информацию);
- **Generate no warnings** (Не сообщать об ошибках);
- **Additional command line parameters** (Дополнительные параметры командной строки).

4.2.3. Диалог настройки *Version Control (Контроль версий)*

С помощью диалога настройки системы контроля версий можно связать текущий проект с той или иной системой контроля версий (рис. 4.88).

В базовой поставке IntelliJ IDEA имеется поддержка трех систем контроля версий:

- **SourceSafe** (*см. раздел 12.4*);
- **CVS** (*см. раздел 12.3*);
- **StarTeam** (*см. раздел 12.5*).

Выбрав одну из систем контроля версий из выпадающего списка, можно настроить ее опции для проекта. Одновременно можно использовать лишь одну систему контроля версий для конкретного проекта.

Системы контроля версий и их настройки подробно рассмотрены в *главе 12*.

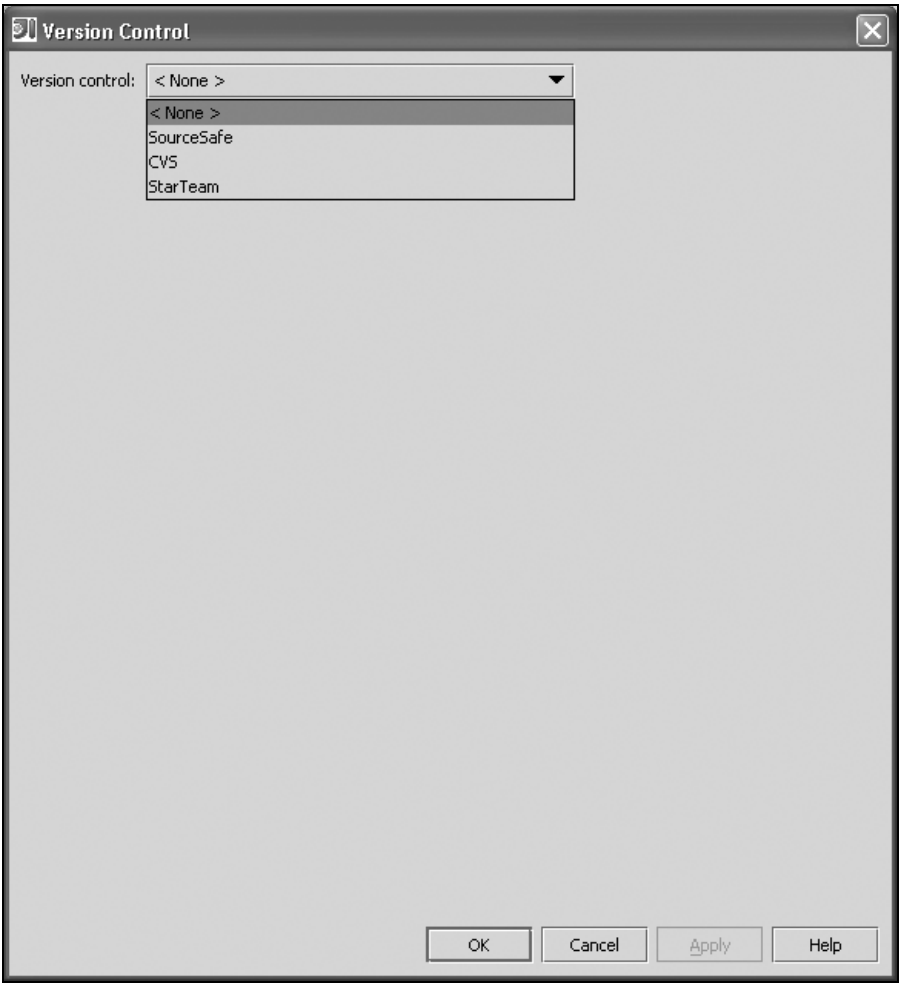


Рис. 4.88. Диалог настройки системы контроля версий для проекта

4.2.4. Диалог настройки *Project Code Style (Стиль кода проекта)*

Каждый проект в IntelliJ IDEA использует по умолчанию глобальные стилевые правила (*см. раздел 4.1.7*). Однако имеется возможность персонально переопределять эти стилевые настройки для проекта.

Прежде чем использовать проектные стилевые настройки, необходимо включить их в диалоге настройки стилей кода для проекта (рис. 4.89).

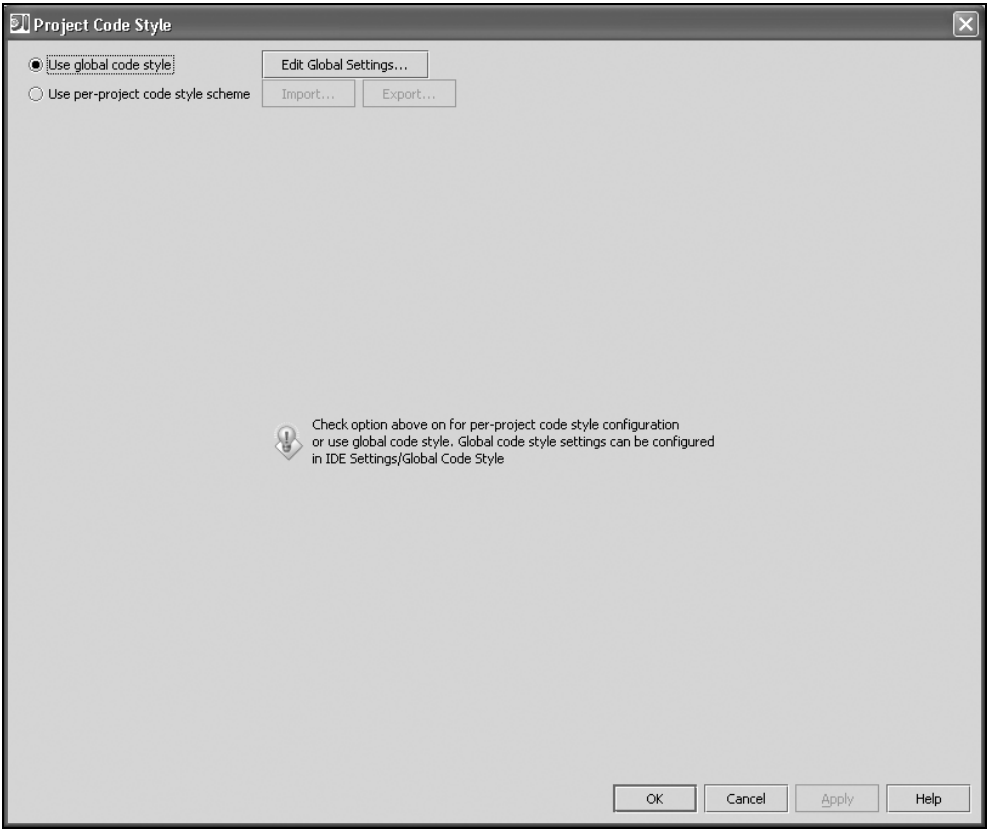


Рис. 4.89. Диалог настройки стилей кода для проекта: персональные стили проекта выключены

Если включить опцию **Use per-project code style scheme** (Включить проектную схему стиля кода), то диалог перейдет в активный режим (рис. 4.90).

Данный диалог настроек идентичен диалогу глобальных настроек стилей кода (*см. раздел 4.1.7*), за исключением двух дополнительных кнопок, расположенных в верхней части диалога:

- **Import** (Импорт) — позволяет импортировать глобальные настройки стилей;
- **Export** (Экспорт) — позволяет экспортировать проектные настройки стилей в глобальные.

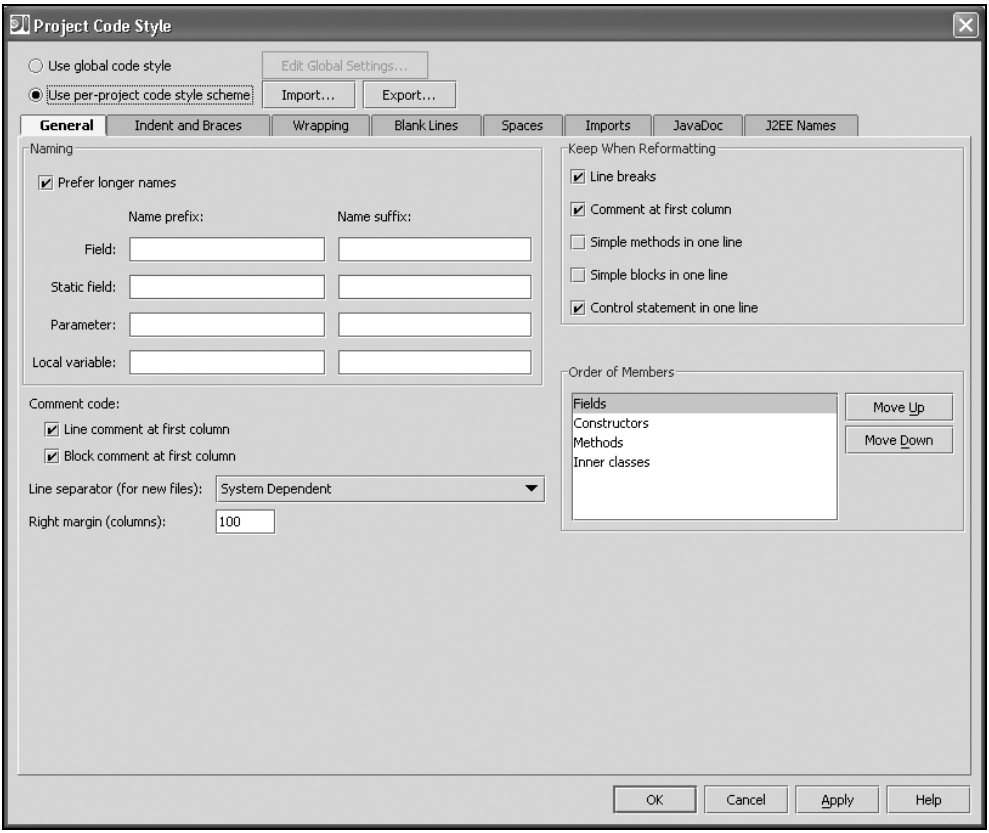
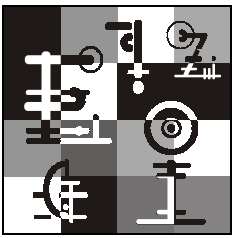


Рис. 4.90. Диалог настройки стилей кода для проекта: персональные стили проекта включены

4.2.5. GUI Designer

Настройки визуального редактора пользовательского интерфейса будут подробно рассмотрены в *главе 7*.



Часть II

Создание кода

Глава 5. Управление проектами

Глава 6. Работа с кодом

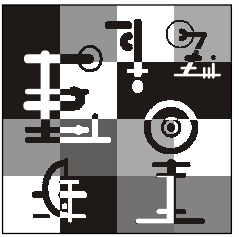
Глава 7. Создание интерфейса пользователя

Глава 8. Рефакторинг

Глава 9. Компиляция, запуск, отладка

Глава 10. J2EE

Глава 11. Поддержка новых возможностей Java 5



Глава 5

Управление проектами

Основой модели разработки программ в IntelliJ IDEA является *проект* (Project). Проект — это некоторый набор файлов (классов, ресурсов, JSP-страниц и пр.) и настроек, актуальных для всех этих файлов. Все файлы в проекте объединены в *модули*. Модуль — это структурная единица проекта, которая позволяет объединять файлы проекта некоторым логическим образом, а также определить для них некоторые специфичные настройки, отличающиеся от общих настроек проекта.

Файлы проектов и модулей записаны в XML-формате и имеют расширения ipr (ipr — IDEA Project) и iml (iml — IDEA Module).

5.1. Структура проекта

Проекты в IntelliJ IDEA имеют иерархическую структуру, в корне которой находится сам проектный файл. Его ближайшими дочерними узлами являются модули, а также библиотеки проекта. Модуль задает множество директорий, которые содержат файлы модуля — так называемые Content Roots (Корневые директории с контентом), а также библиотеки модуля.

Физическое расположение директорий и файлов проекта может отличаться от иерархической структуры, задающей логическую модель.

Более подробно структуру проекта и его модулей мы рассмотрим в *разделе 5.3*.

5.2. Создание проекта

В данном разделе мы рассмотрим вариант создания абсолютно нового проекта.

Для создания проекта используется команда меню **File | New Project** (Файл | Новый проект), которая запускает мастер создания проектов (рис. 5.1).

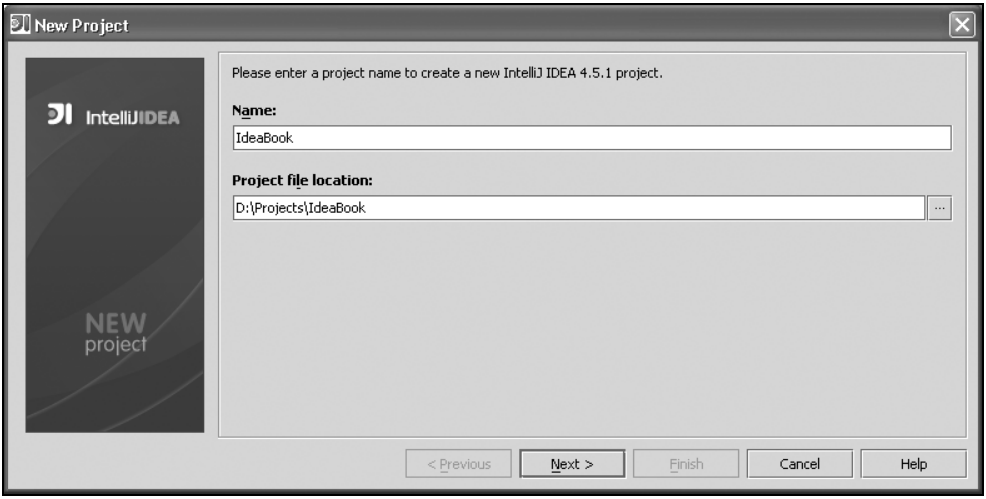


Рис. 5.1. Запуск мастера создания проектов

В мастере можно указать директорию проекта — ту, в которой будет находиться файл проекта (**Project file location**), а также его имя (**Name**) — имя файла проекта.

Если выбранная директория не существует, IntelliJ IDEA предложит ее создать.

5.2.1. Выбор JDK

Затем IntelliJ IDEA предложит выбрать JDK для проекта. Выбрать можно из списка доступных JDK (*см. раздел 4.1.8*) (рис. 5.2).

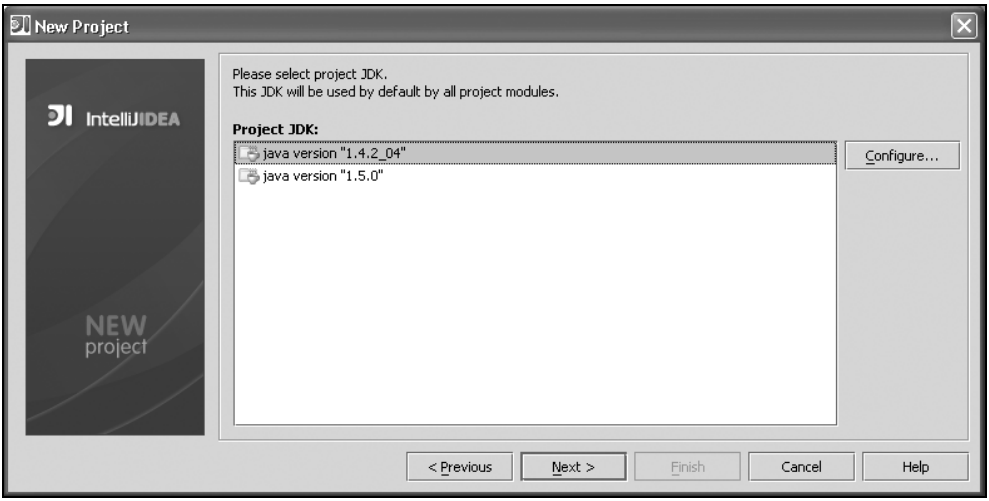


Рис. 5.2. Выбор JDK для проекта

Если в списке нет требуемого JDK, с помощью кнопки **Configure** (Конфигурировать) можно открыть диалог настройки и сконфигурировать требуемый JDK (*см. раздел 4.1.8*).

5.2.2. Выбор количества модулей

На следующем шаге IntelliJ IDEA предлагает выбрать один из двух вариантов (рис. 5.3):

- **Create single-module project** (Создать проект с одним модулем);
- **Create/configure multi-module project** (Создать проект с несколькими модулями).

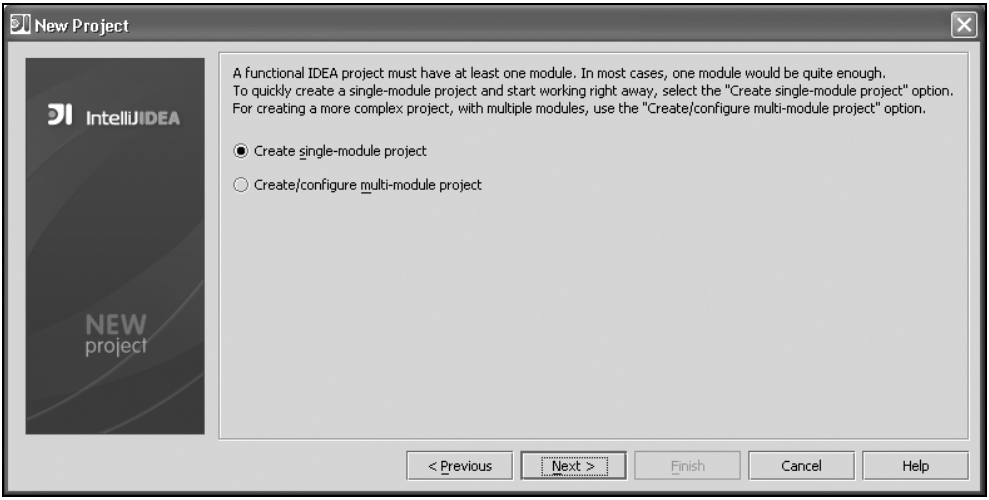


Рис. 5.3. Выбор между проектом с одним модулем и несколькими

В примечании к данной странице мастера настроек сказано: "Проект IntelliJ IDEA с минимальной функциональностью должен иметь как минимум один модуль. В большинстве случаев одного модуля бывает достаточно".

Надо заметить, что в проекте может и не быть модулей, однако, в этом случае, в проекте не может быть и файлов.

Для начала нам хватит одного модуля, так что выберем проект с одним модулем.

5.2.3. Выбор типа модуля

На следующем шаге IntelliJ IDEA предложит выбрать тип модуля для создания вместе с проектом (рис. 5.4).

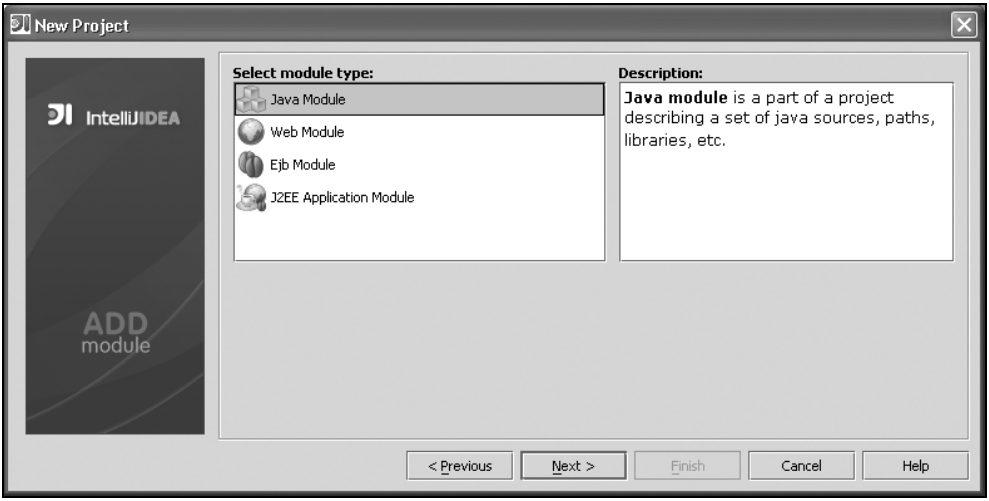


Рис. 5.4. Выбор типа модуля для создания с проектом

Есть четыре варианта:

- **Java Module** (Java-модуль) — самый простой вариант модуля. Java-модуль — это модуль, объединяющий произвольные Java-файлы;
- **Web Module** (Web-модуль) — модуль, содержащий файл web.xml (Web deployment descriptor — XML-дескриптор Web-приложения), в котором содержатся ссылки на сервлеты (servlets), фильтры (filters), JSP-файлы, являющиеся составной частью Web-приложения;
- **Ejb Module** (EJB-модуль) — модуль, содержащий файл ejb-jar.xml (EJB deployment descriptor — XML-дескриптор EJB), в котором содержатся ссылки на все EJB;
- **J2EE Application Module** (Модуль J2EE-приложения) — данный модуль содержит другие J2EE-модули (Web-модули и EJB-модули) в архиве J2EE-приложения (EAR — Enterprise Archive), используя стандартный XML-дескриптор J2EE-приложения (application.xml).

Рассмотрим варианты создания всех типов модулей.

5.2.3.1. Создание Java-модуля

Выбор места расположения модуля

Выбрав тип Java-модуля, мы перейдем к следующей странице мастера создания проекта, а именно к выбору названия и месту расположения файла модуля (рис. 5.5).

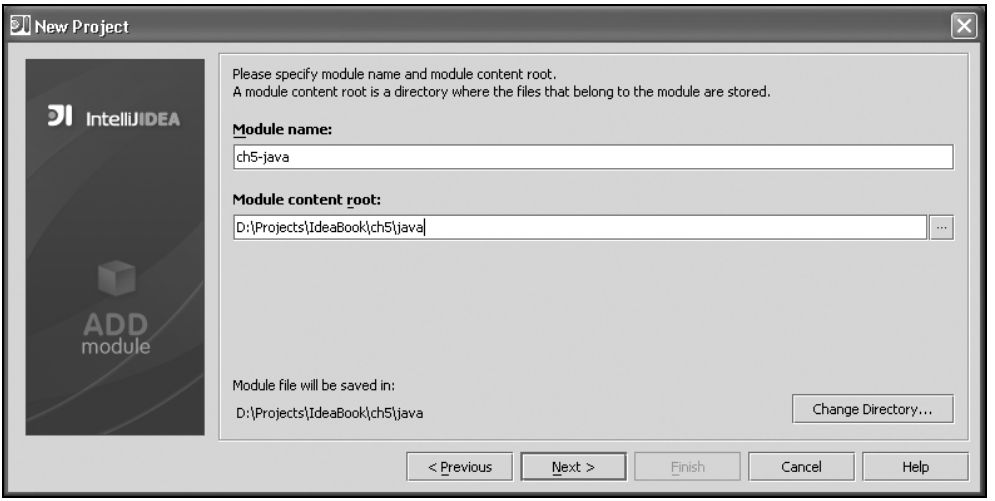


Рис. 5.5. Выбор названия модуля и место расположения его файла

Создание директории для исходных файлов

На следующем шаге нужно выбрать, создавать или нет директорию для исходных файлов (source path) (рис. 5.6).

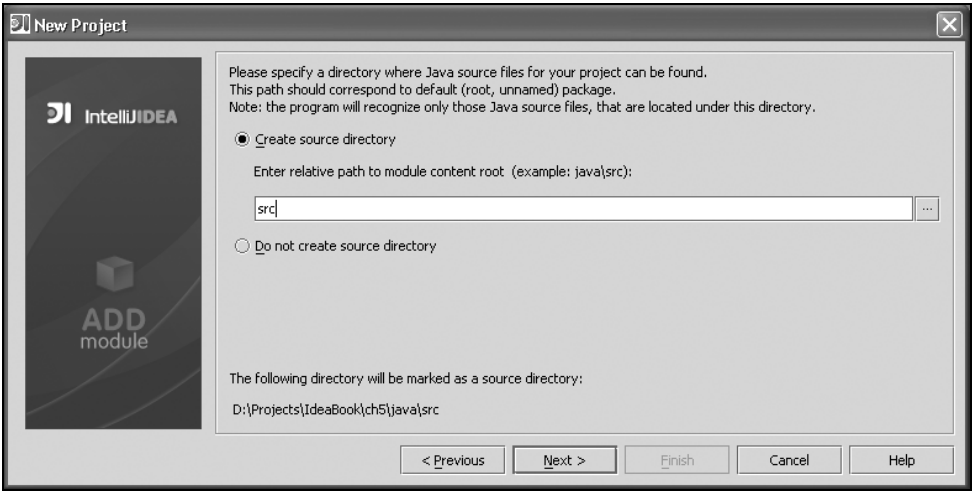


Рис. 5.6. Создание директории исходных файлов модуля

IntelliJ IDEA может работать с Java-классами, только если их файлы находятся в директории для исходных файлов (специально помеченной директории). Имеется возможность не создавать директорию. В дальнейшем ее можно будет добавить при настройке модуля.

Мы будем использовать название "src", выбранное по умолчанию.

Выбор выходной директории

На последнем шаге нужно выбрать директорию, в которой будут размещаться скомпилированные файлы Java-классов (рис. 5.7).

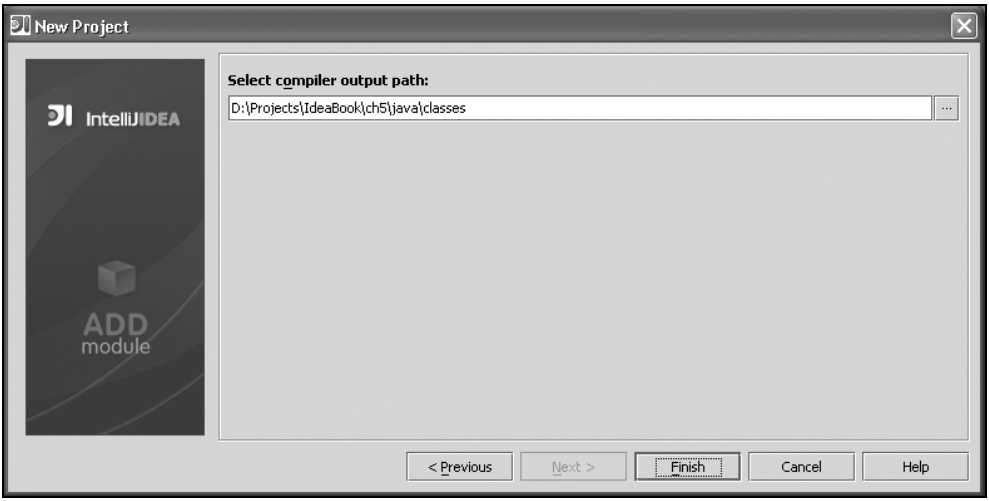


Рис. 5.7. Выбор выходной директории

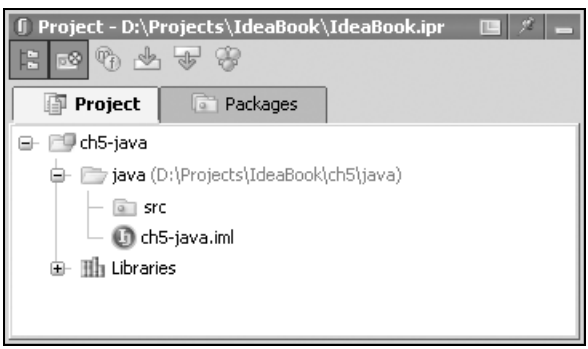


Рис. 5.8. Структура нового проекта

Наш проект создан (рис. 5.8).

На рис. 5.8 представлена структура вновь созданного проекта в инструментальном окне **Project** (Проект) (*см. раздел 3.4.8.2*).

После того, как мы создали готовый одномодульный проект, можно добавить в него дополнительные модули. Мы будем добавлять новые модули в существующий проект. То, что будет описано в следующем разделе, ничем не отличается от ситуации, когда создается проект с отдельным Web-, EJB- или J2EE Application-модулем.

5.2.3.2. Создание Web-модуля

Прежде, чем рассказать о создании Web-модуля, сделаем небольшое замечание. Та структура каталогов размещения файлов Web-модуля, которую по умолчанию выбирает IntelliJ IDEA, не является оптимальной и часто может приводить к неработающим конфигурациям. Поэтому, если вы создаете новый Web-модуль, мы рекомендуем вам использовать следующую структуру директорий для него:

```
Project/ - директория проекта
        Web-module/ - директория Web-модуля
                    resources/ - директория для ресурсов
                        WEB-INF/
                                web.xml
```

Отличие данной структуры от структуры по умолчанию в том, что XML-дескриптор Web-модуля размещен вместе с папкой WEB-INF внутри директории с ресурсами (там будет расположено содержимое Web-приложения: файлы HTML, JSP и т. п.).

Такое размещение файлов дает дополнительное преимущество — можно использовать директорию ресурсов в качестве рабочей директории Web-приложения, т. е. не нужно будет тратить время на копирование файлов Web-приложения в другую рабочую директорию (deployment).

Если вам все-таки требуется разделить исходную директорию с ресурсами Web-модуля и рабочую директорию, можно использовать такую структуру:

```
Project/ - директория проекта
        Web-module/ - директория Web-модуля
                    resources/ - директория для ресурсов
                        WEB-INF/
                                web.xml
                    exploded/ - рабочая директория
                        WEB-INF/
                                classes/
```

Выбор места расположения модуля

Данный шаг идентичен выбору места размещения Java-модуля (рис. 5.9).

Выбор сервера приложений

На следующем шаге IntelliJ IDEA предложит нам выбрать сервер приложений из списка доступных серверов (рис. 5.10). Если ни одного сервера не сконфигурировано, с помощью диалога настройки серверов приложений можно их настроить (*см. раздел 4.1.20*).

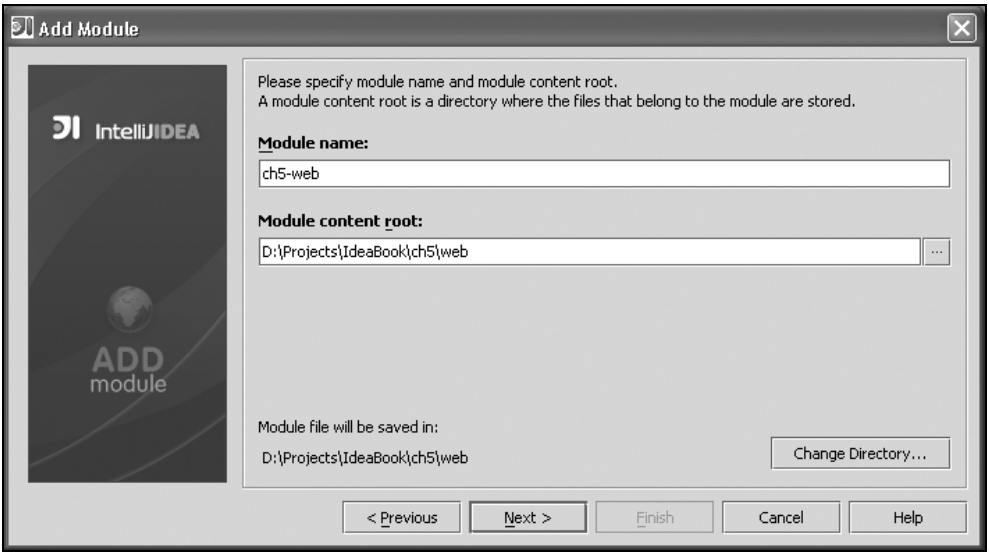


Рис. 5.9. Выбор места размещения Web-модуля

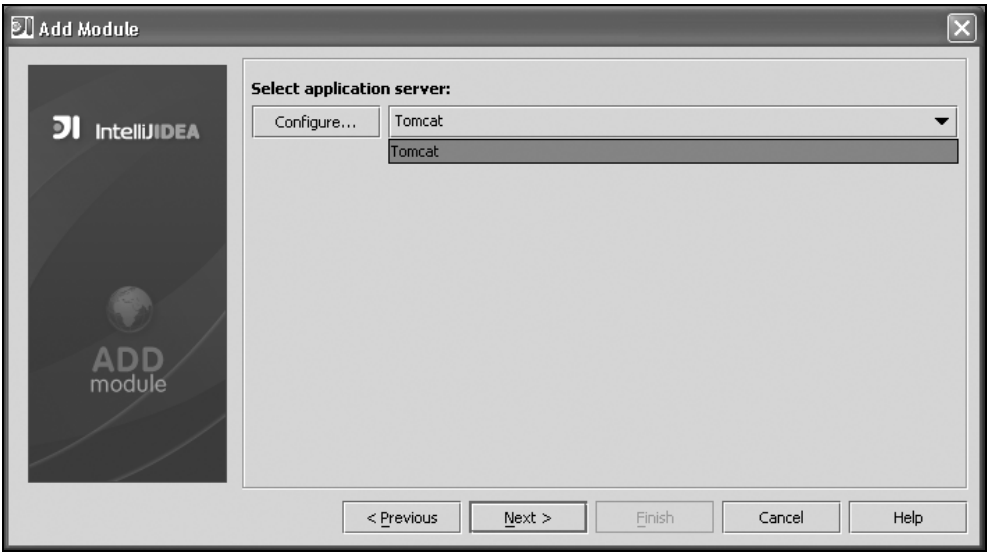


Рис. 5.10. Выбор сервера приложений для Web-модуля

В принципе, этот шаг можно пропустить и настроить сервер приложений позже. Однако забывать об этом не стоит, поскольку иначе Web-приложение не может быть запущено.

Выбор места размещения XML-дескриптора

Для корректной работы Web-модуля необходим XML-дескриптор Web-модуля. Если мы создаем новый Web-модуль на "пустом месте", IntelliJ IDEA может создать такой XML-дескриптор (рис. 5.11).

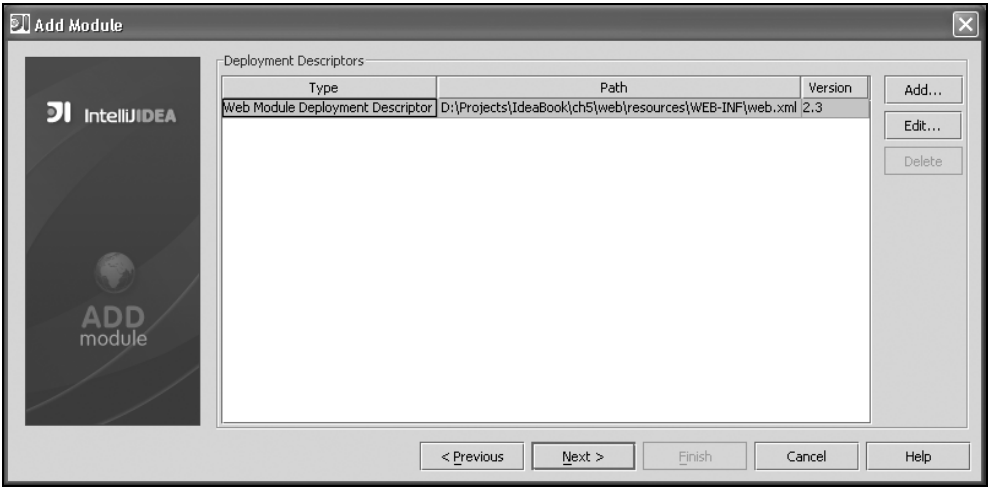


Рис. 5.11. Выбор XML-дескриптора

Изменить место расположения XML-дескриптора можно с помощью кнопки **Edit** (Редактировать).

В открывшемся диалоге (рис. 5.12) также можно выбрать версию XML-дескриптора:

- **2.2**;
- **2.3**;
- **2.4**.

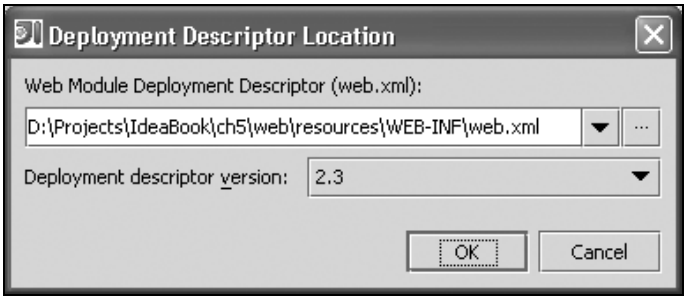


Рис. 5.12. Выбор места размещения XML-дескриптора

Примечание

Если вы используете сервер WebLogic, набор дескрипторов может отличаться от приведенного, поскольку WebLogic требует наличия собственного дескриптора.

Выбор директории ресурсов

Далее следует этап выбора директории с файлами ресурсов Web-модуля (рис. 5.13).

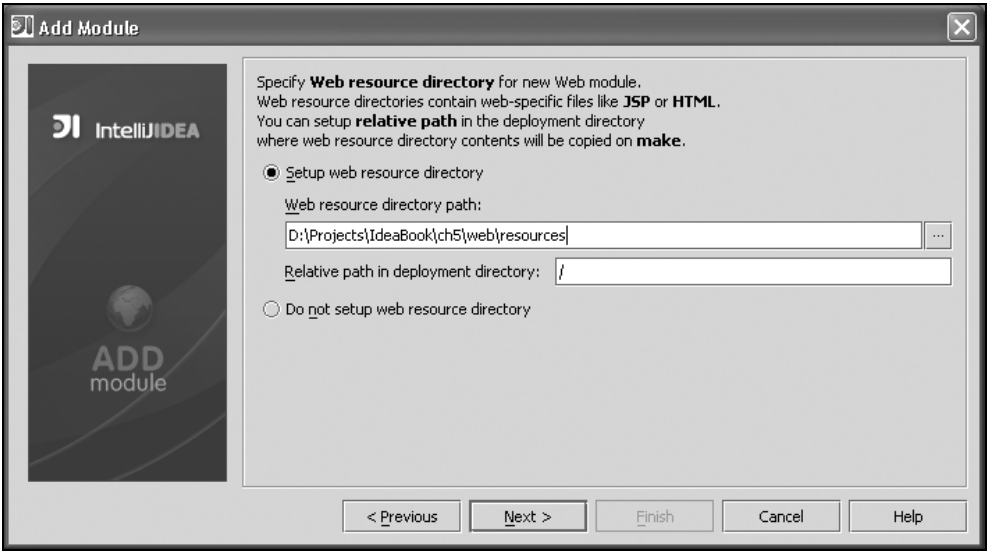


Рис. 5.13. Выбор директории ресурсов Web-модуля

В директории ресурсов Web-модуля размещаются файлы, доступные для данного Web-приложения: HTML, JSP и т. п.

Кроме самой директории ресурсов (**Web resource directory path**) можно задать путь для данных ресурсов относительно рабочей директории Web-приложения (**Relative path in deployment directory**). По умолчанию используется корневая директория.

Также имеется возможность не настраивать директорию с ресурсами (**Do not setup web resource directory**).

Выбор рабочей директории

Далее следует шаг выбора рабочей директории для Web-приложения. Как мы уже советовали в начале этого раздела, рабочая директория может совпадать с директорией ресурсов, если соответствующим образом настроить взаимное расположение директорий Web-модуля.

Рабочая директория нашего Web-приложения будет совпадать с директорией ресурсов (рис. 5.14).

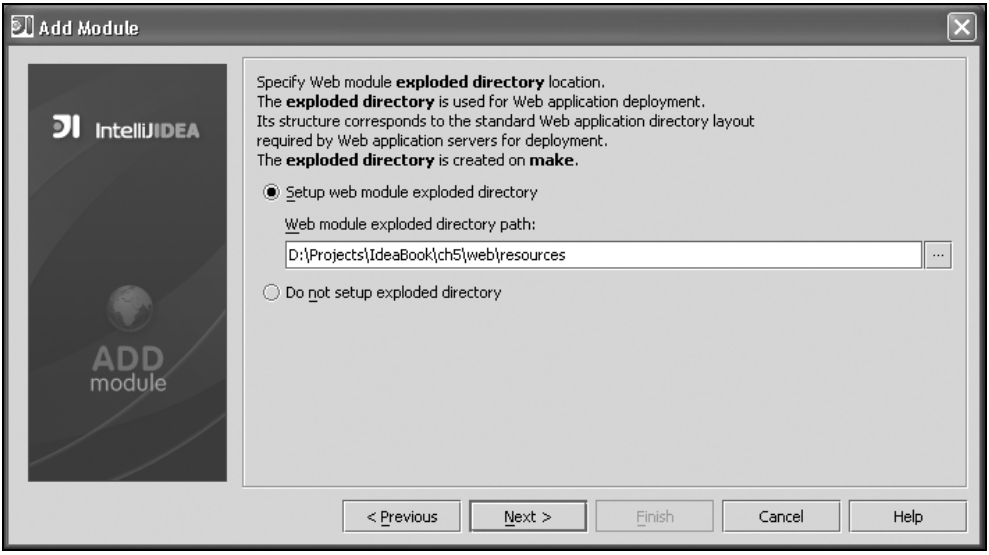


Рис. 5.14. Выбор рабочей директории Web-приложения

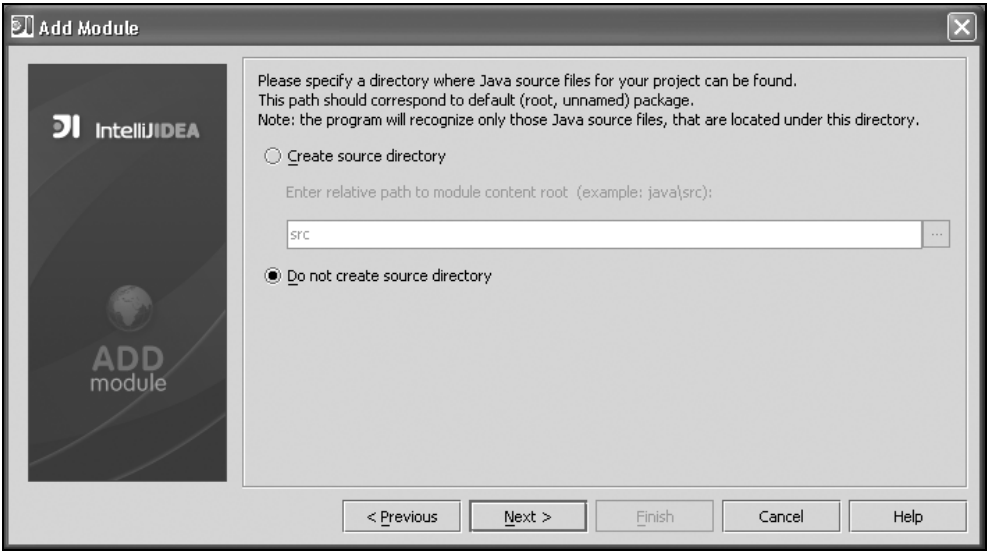


Рис. 5.15. Выбор директории для исходных файлов

Выбор директории для исходных файлов

На следующем шаге мастер создания Web-модуля предложит выбрать место размещения директории для исходных файлов. Обратите внимание, имеются в виду Java-файлы, а не JSP. Если Web-приложение не содержит Java-файлов, эту директорию можно не создавать, либо создать ее позже.

Если эти файлы есть, директорию можно также не создавать, а вынести Java-файлы в отдельный модуль, классы которого могут затем собираться в отдельный JAR-файл, а он уже будет входить в Web-приложение как библиотека. Впрочем, тут все зависит от проекта.

Наше Web-приложение не будет содержать Java-файлов, поэтому мы не будет создавать директорию для исходных файлов (рис. 5.15).

Выбор выходной директории

На последнем шаге мастер создания Web-приложения попросит нас указать выходную директорию для нашего Web-модуля (рис. 5.16).

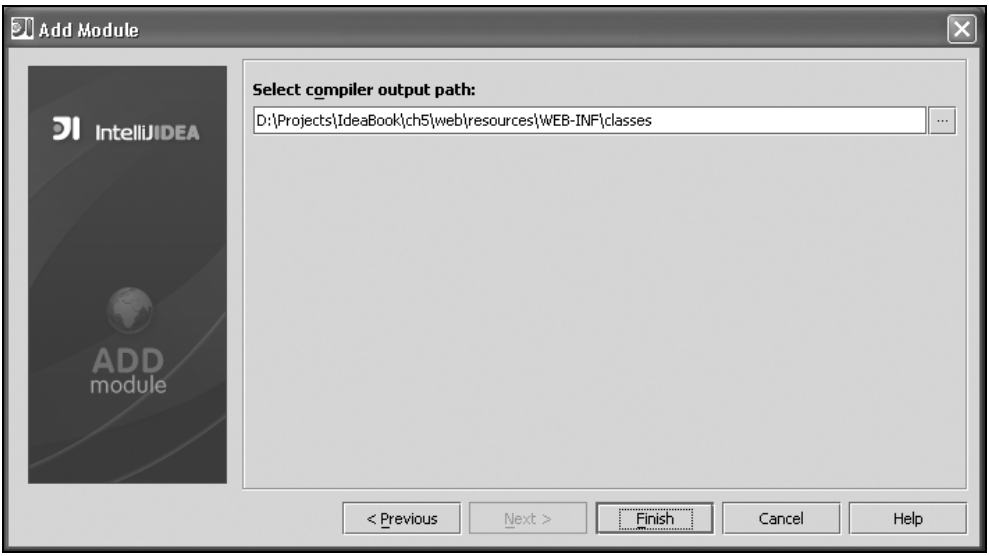


Рис. 5.16. Выбор выходной директории

Если Web-приложение не содержит Java-файлов, эту директорию, в принципе, можно было бы и не задавать, т. к. компилировать в нее все равно будет нечего. Однако IntelliJ IDEA требуется наличие выходной директории для каждого модуля.

Итак, наш Web-модуль создан (рис. 5.17).

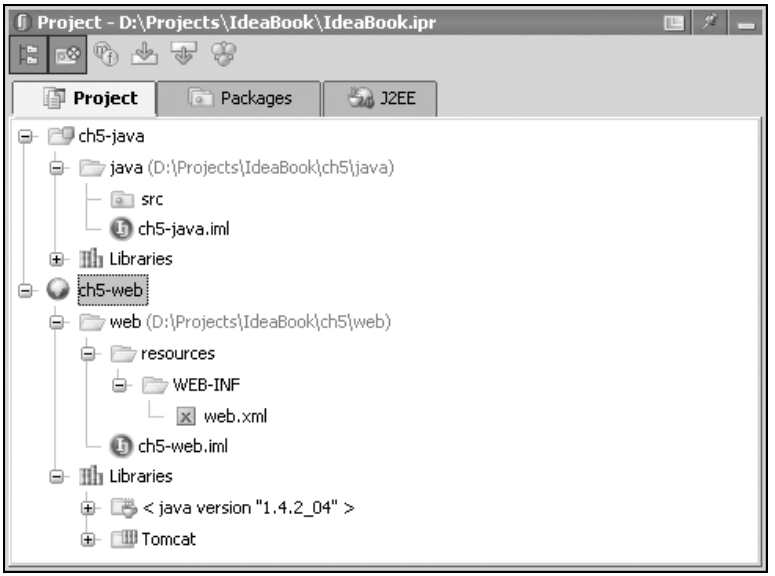


Рис. 5.17. Web-модуль в структуре проекта

Обратите внимание на новую закладку **J2EE**, появившуюся в инструментальном окне **Project** (Проект). В этой закладке отображается более детализированная структура Web-модулей (а также EJB-модулей и J2EE Application-модулей) (рис. 5.18).

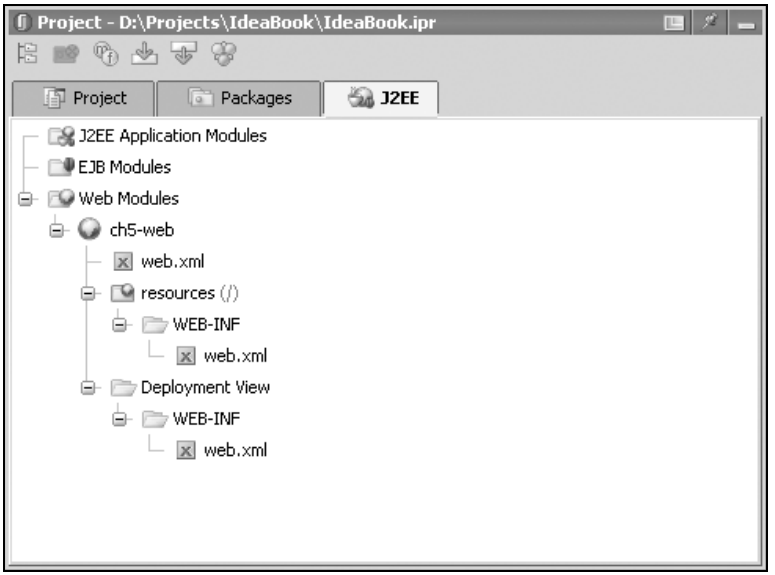


Рис. 5.18. Содержание закладки **J2EE** инструментального окна **Project**

Если вы дословно повторяли наши шаги, а добиться такой же картинки вам не удалось, — не беда. Есть одна небольшая хитрость, о которой мы еще не успели рассказать.

По умолчанию из просмотра дерева проекта IntelliJ IDEA исключает директории, предназначенные для размещения рабочих файлов: скомпилированных классов и т. п. Для Web-модуля такой директорией является рабочая директория Web-приложения, которая в случае нашего проекта совпадает с директорией ресурсов.

Чтобы структура отобразилась правильно, нужно в диалоге настройки путей проекта (*см. раздел 5.3.1*) выключить опцию **Excluded** (Исключенные) для директории с ресурсами (рис. 5.19).

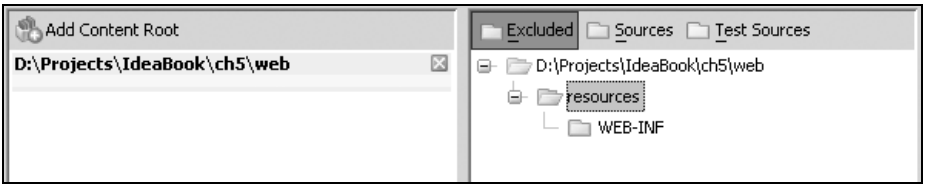


Рис. 5.19. Опция **Excluded** выключена для директории ресурсов Web-модуля

5.2.3.3. Создание EJB-модуля

Выполним команду создания модуля и выберем тип модуля EJB-модуль. Затем в мастере создания модуля укажем название модуля и директорию, в которой будет находиться файл модуля (рис. 5.20).

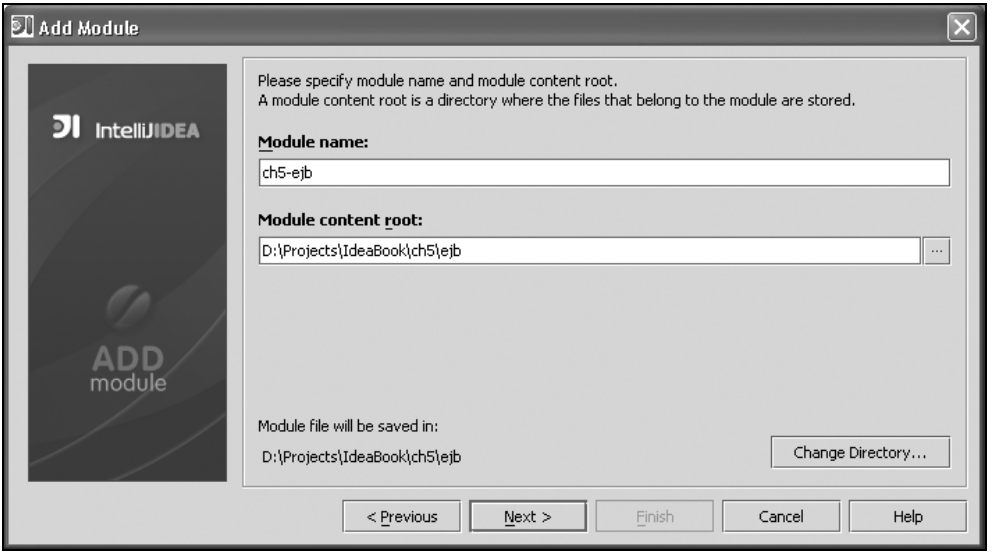


Рис. 5.20. Выбор названия модуля и директории размещения файла модуля

На следующем шаге нужно указать название директории с исходными файлами (рис. 5.21).

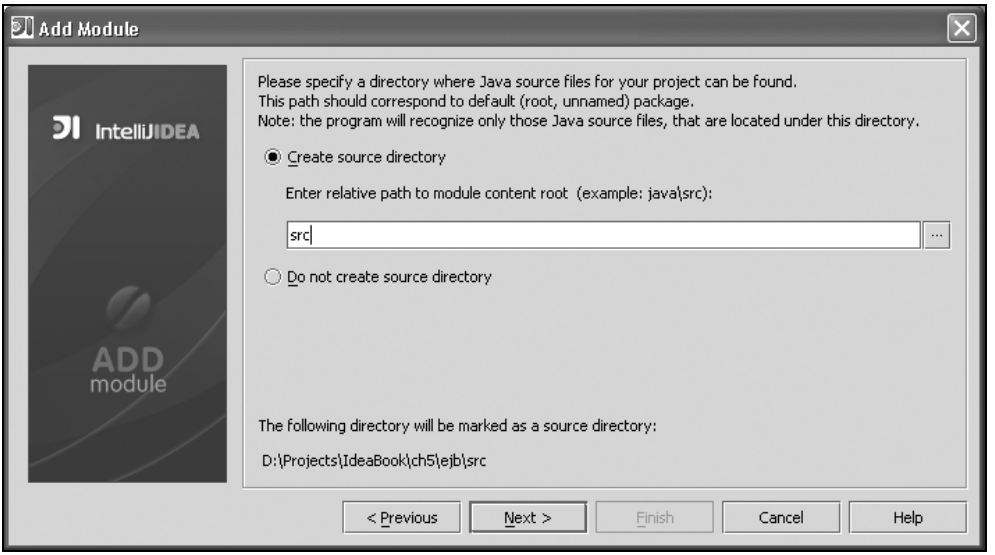


Рис. 5.21. Выбор директории с исходными файлами модуля

Далее нужно выбрать место размещения скомпилированных классов — выходную директорию (рис. 5.22).

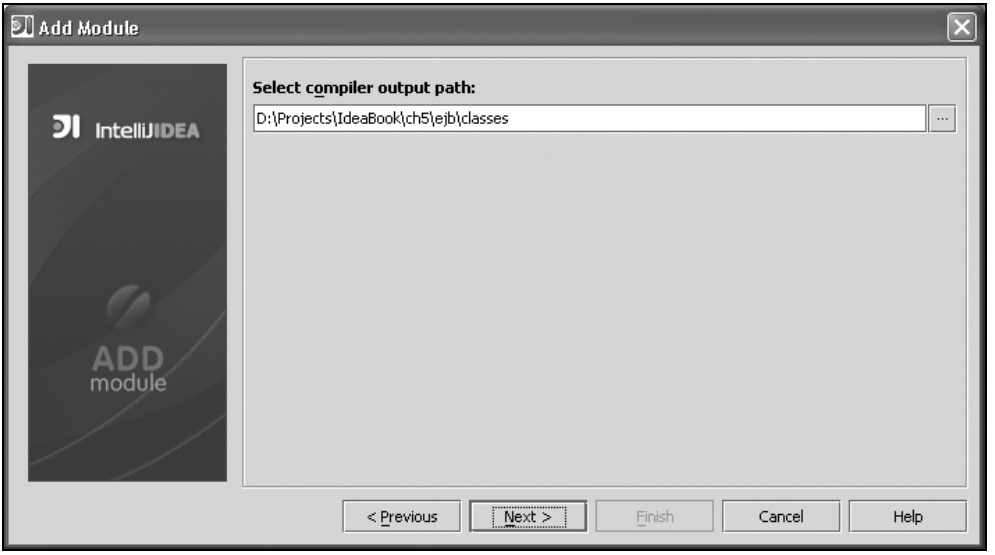


Рис. 5.22. Выбор выходной директории модуля

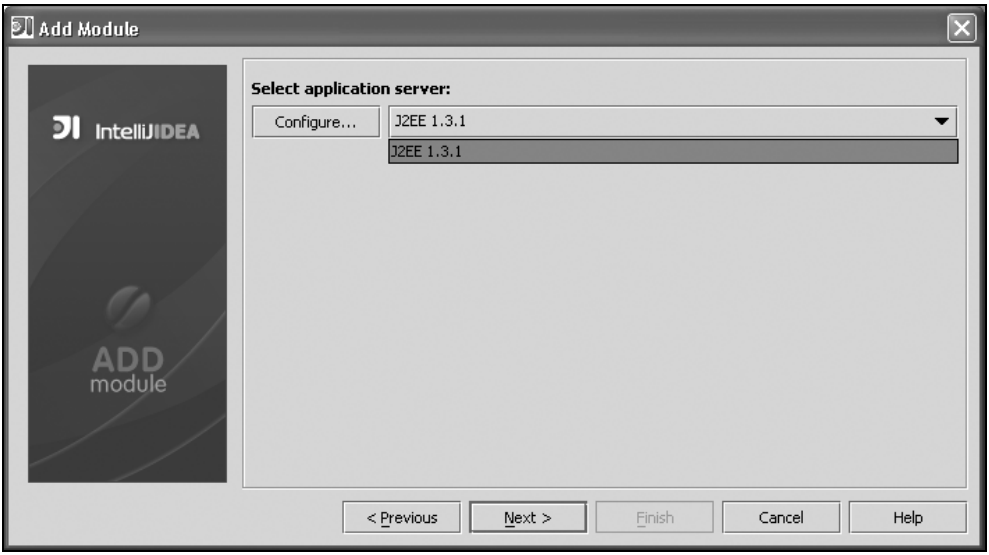


Рис. 5.23. Выбор сервера приложений для EJB-модуля

Затем нужно выбрать сервер приложений для EJB-модуля (рис. 5.23).

В списке доступных серверов будут отображены только те серверы, которые поддерживают EJB. В случае нашего модуля — это сервер приложений из поставки J2EE 1.3.1, настройку которого мы рассматривали в *разделе 4.1.20.3*.

На следующем шаге нужно указать все доступные XML-дескрипторы развертывания EJB — ejb-jar.xml (рис. 5.24).

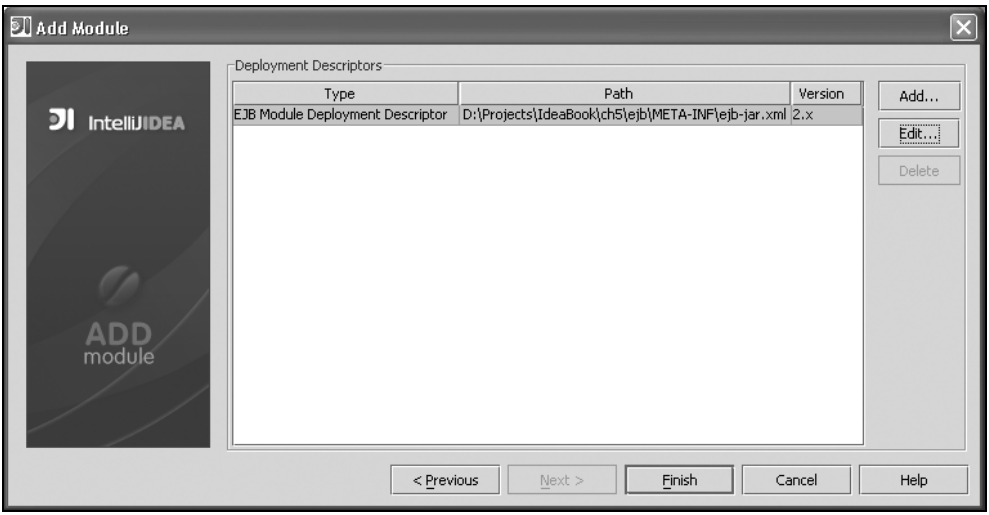


Рис. 5.24. Настройка XML-дескрипторов EJB

По умолчанию задается XML-дескриптор, расположенный в директории META-INF нашего модуля. Добавить новый дескриптор (кнопка **Add**) и отредактировать имеющиеся (кнопка **Edit**) можно с помощью диалога, представленного на рис. 5.25.

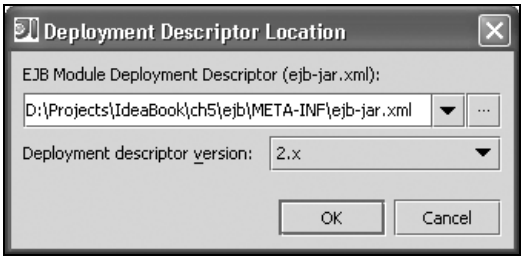


Рис. 5.25. Диалог редактирования свойств XML-дескриптора

Данный диалог позволяет настроить два параметра:

- место расположения дескриптора (**EJB Module Deployment Descriptor**);
- версию дескриптора (**Deployment descriptor version**):
  - **2.x**;
  - **1.x**.

Дескрипторы, указанные на данном шаге, будут автоматически созданы IntelliJ IDEA.

Примечание

Если вы используете сервер WebLogic, набор дескрипторов может отличаться от приведенного, поскольку WebLogic требует наличия собственного дескриптора.

После подтверждения работы мастера создания модуля наш модуль будет создан (рис. 5.26).

Рис. 5.26. EJB-модуль создан

5.2.3.4. Создание J2EE Application-модуля

Вызовем с помощью команды создания нового модуля мастер создания модулей и выберем тип модуля **J2EE Application Module**. На первом шаге нам понадобится указать место размещения файла модуля, а также его имя (рис. 5.27).

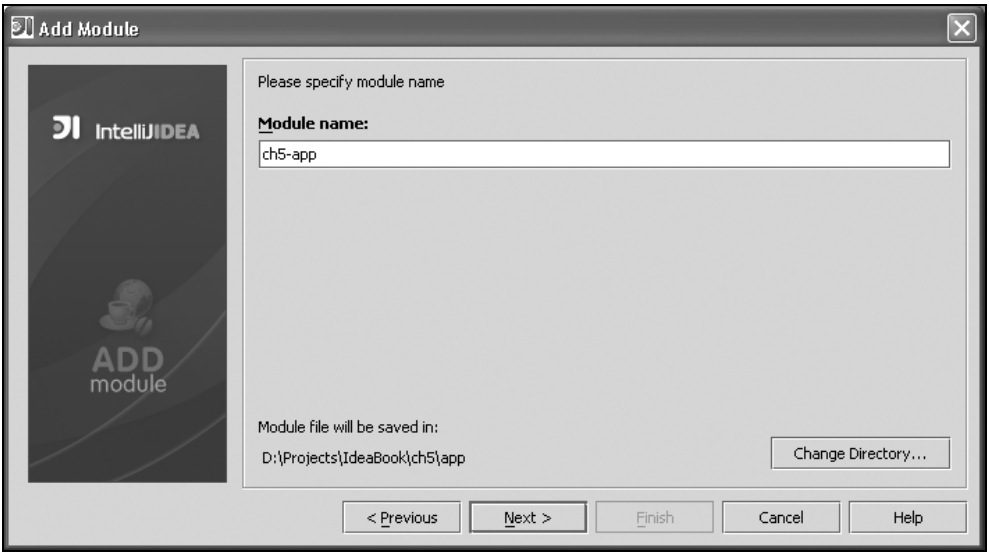


Рис. 5.27. Выбор имени и места размещения модуля

Место размещения модуля можно настроить с помощью диалога выбора пути, открывающегося с помощью кнопки **Change Directory** (Поменять директорию).

Далее следует выбрать сервер приложений для использования с нашим модулем (рис. 5.28).

В списке доступных серверов приложений будут присутствовать только те серверы, которые поддерживают J2EE-приложения. В случае нашего модуля мы будет использовать сервер приложений, создание которого рассматривали в *разделе 4.1.20.3.*

На следующем шаге можно задать XML-дескриптор развертывания J2EE-приложения — application.xml (рис. 5.29).

Создать новый дескриптор (кнопка **Add**) и отредактировать параметры имеющегося (кнопка **Edit**) дескриптора можно с помощью диалога, представленного на рис. 5.30.

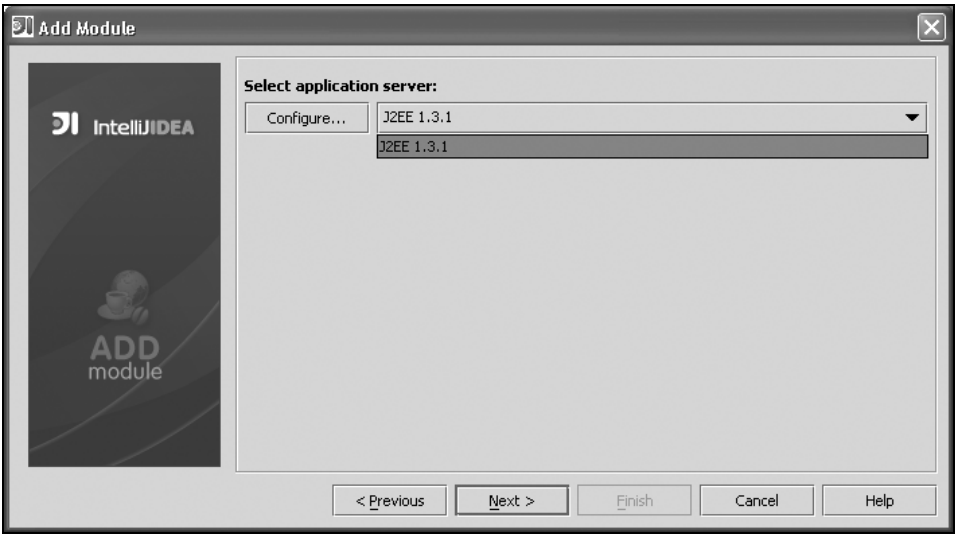


Рис. 5.28. Выбор сервера приложений

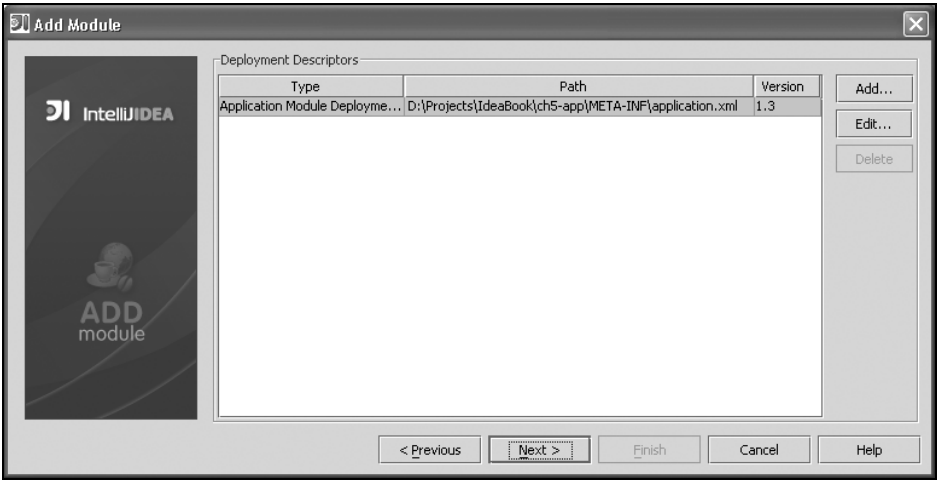


Рис. 5.29. Настройка дескриптора J2EE-приложения

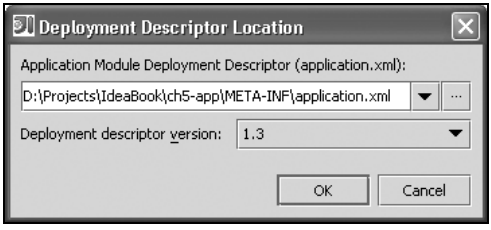


Рис. 5.30. Настройка параметров дескриптора J2EE-приложения

С помощью данного диалога можно указать:

- путь, по которому будет создан дескриптор;
- а также его версию (**Deployment descriptor version**):
  - **1.2**;
  - **1.3**;
  - **1.4**.

Примечание

Если вы используете сервер WebLogic, набор дескрипторов может отличаться от приведенного, поскольку WebLogic требует наличия собственного дескриптора.

На заключительном шаге создания модуля J2EE-приложения необходимо выбрать те модули проекта, которые будут включены в J2EE-приложение (рис. 5.31).

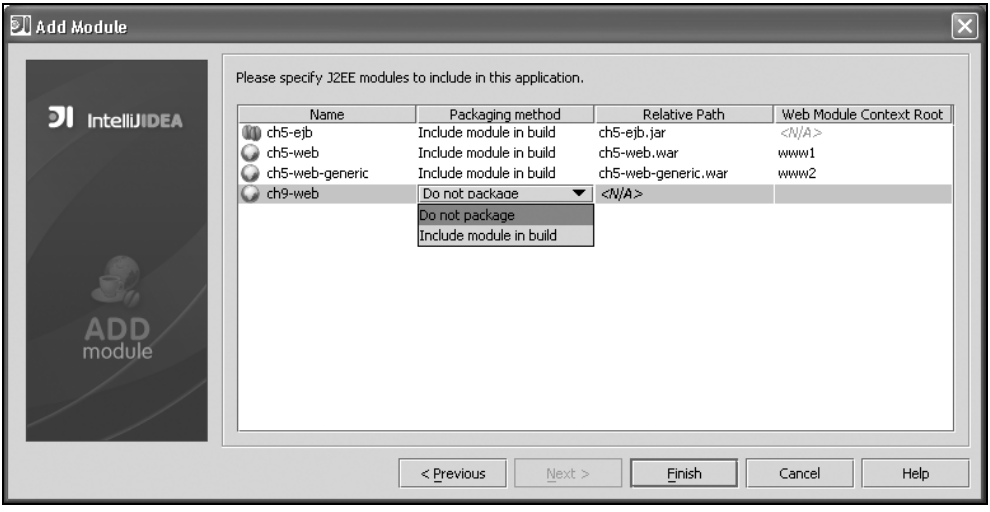


Рис. 5.31. Настройка модулей проекта, которые будут включены в J2EE-приложение

В список доступных модулей включены все Web- и EJB-модули, имеющиеся в проекте. Для каждого модуля можно указать:

- будет ли он включен в J2EE-приложение (**Include module in build**) или нет (**Do not package**);
- относительный путь (**Relative Path**);
- а для Web-модулей также можно указать контекстный путь (**Web Module Context Root**), относительно которого он будет доступен в J2EE-приложении.

После подтверждения мастера создания модуля J2EE-модуль будет создан (рис. 5.32).

Рис. 5.32. Модуль J2EE-приложения создан

5.3. Редактирование свойств модуля

Свойства модулей проекта настраиваются с помощью диалога **Paths** (Пути) из настроек проекта (*см. раздел 4.2*).

Общий вид диалога настроек путей представлен на рис. 5.33.

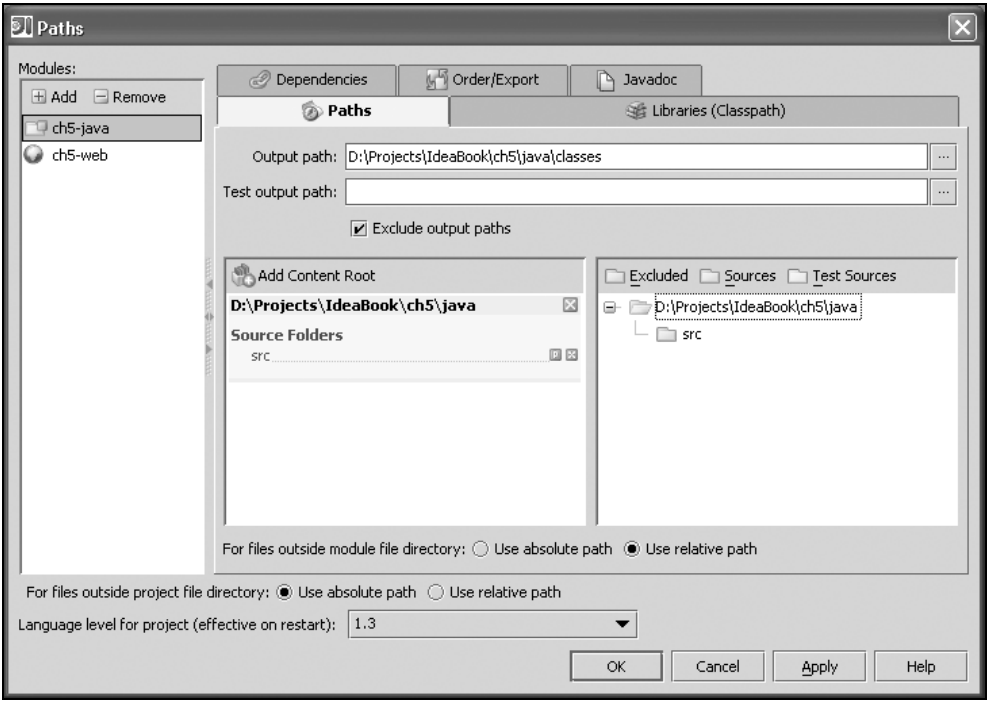


Рис. 5.33. Свойства модулей в диалоге **Paths**

В левой части диалога расположен список модулей, имеющихся в проекте. Над списком расположены две управляющие кнопки для добавления модуля в проект (при этом запустится мастер создания модулей (*см. раздел 5.2.3*)).

Левую часть диалога занимают настройки выбранного модуля, разделенные на несколько закладок. Для Java-модулей это следующие закладки:

- **Paths** (Пути) — на данной закладке настраиваются пути всех директорий, относящихся к модулю;
- **Libraries (Classpath)** (Библиотеки) — на данной закладке можно настроить библиотеки модуля, а также выбрать персональный JDK, если это требуется;
- **Dependencies** (Зависимости) — данная закладка содержит список модулей, от которых зависит текущий модуль;
- **Order/Export** (Порядок/Экспорт) — на данной закладке указывается порядок загрузки библиотек модуля, а также возможность экспорта зависимых модулей (подробнее об экспорте будет рассказано в *разделе 5.3.5*);
- **Javadoc** (комментарии JavaDoc) — на данной закладке можно задать перечень JavaDoc-комментариев для использования совместно с классами модуля.

У других модулей набор закладок может быть иным. Так, например, у Web-модулей дополнительно имеются две закладки:

- **Web Module Settings** (Настройки Web-модуля);
- **J2EE Build Settings** (Настройки сборки J2EE).

У EJB-модуля:

- **Ejb Module Settings** (Свойства EJB-модуля);
- **J2EE Build Settings** (Свойства сборки J2EE).

У J2EE Application-модуля:

- **J2EE Application Module Settings** (Свойства J2EE Application-модуля);
- **J2EE Build Settings** (Свойства сборки J2EE).

Причем у J2EE Application-модуля нет других закладок, кроме этих двух.

5.3.1. Закладка *Paths (Пути)*

На закладке **Paths** (Пути) расположены следующие опции (см. рис. 5.33):

- **Output path** (Выходная директория) — директория, в которую будут складываться откомпилированные классы. Не рекомендуется устанавливать ее равной директории с исходными файлами;
- **Test output path** (Выходная директория для тестов) — директория, в которую будут складываться откомпилированные классы тестов;

- **Exclude output path** (Не включать выходные директории) — данная опция позволяет исключить из иерархии проекта, доступной для просмотра в инструментальном окне **Project** (Проект) (*см. раздел 3.4.8.2*) выходные директории. По умолчанию данная опция включена;
- **For files outside project file directory** (Для файлов, расположенных за пределами директории с файлом модуля):
  - **Use absolute path** (Использовать абсолютный путь);
  - **Use relative path** (Использовать относительный путь).

В центральной части диалога содержатся две панели, на левой из которых перечислены корневые директории с контентом (**Content Root**), а на правой — задаются свойства отдельных директорий, относящихся к той или иной корневой директории.

Корневые директории с контентом можно добавлять с помощью кнопки **Add Content Root** (Добавить корневую директорию) (рис. 5.34).

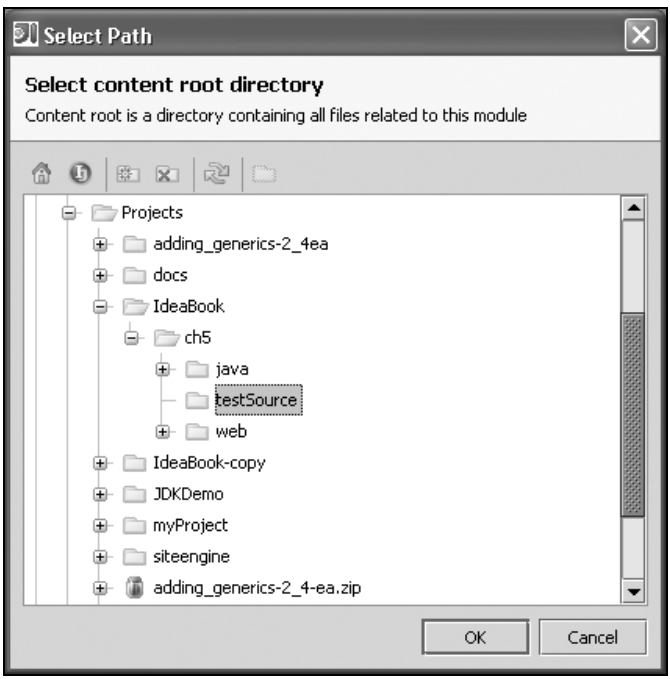


Рис. 5.34. Добавление корневой директории

С помощью кнопок со значком ×, расположенных напротив конкретных корневых директорий, можно удалять из модуля те или иные корневые директории (рис. 5.35).

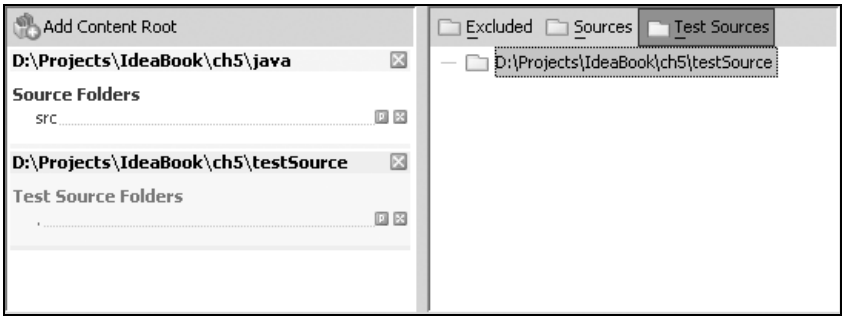


Рис. 5.35. Детальный вид списка корневых директорий

На правой панели перечислены директории, находящиеся под той или иной корневой директорией. Каждая директория может иметь один из трех статусов:

- **Excluded** (Исключена) — директория исключена из обозрения в инструментальном окне **Project** (Проект). Такие директории помечены розоватым цветом;
- **Sources** (Исходные тексты) — директория содержит файлы с исходными текстами на Java. Данные директории помечены голубым цветом;
- **Test Sources** (Исходные тексты тестов) — директория содержит файлы с исходными текстами тестов на Java. Данные директории помечены зеленым цветом.

Цвета на черно-белых иллюстрациях, к сожалению, не видны.

Если директория помечена как содержащая исходные тексты (не важно, тесты там или основные классы), она отображается на левой панели под записью пути к корневой директории. На рис. 5.35 таких директории две — src (под первым корнем) и . (под вторым корнем). Названия директорий даются в относительном значении.

Напротив названий директорий имеется две небольшие кнопки со значками **p** и ×. Кнопка **p** позволяет задать префикс пакета, к которому будут относиться все классы, находящиеся в данной директории (рис. 5.36).

Если префикс не пустой, запись о директории будет дополнена указанием пакета (рис. 5.37).

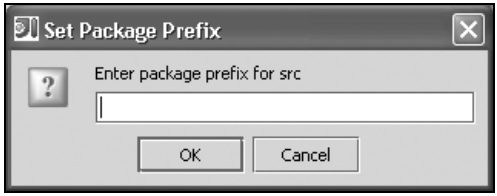


Рис. 5.36. Ввод префикса пакета для директории с исходными текстами

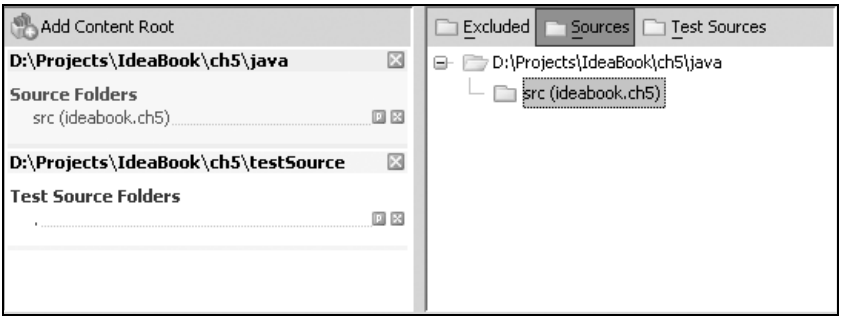


Рис. 5.37. Директории с исходными текстами с заданным префиксом пакета

Примечание

В качестве корневой директории можно задать не только директорию, которая физически присутствует на диске, но директории из архива (рис. 5.38).

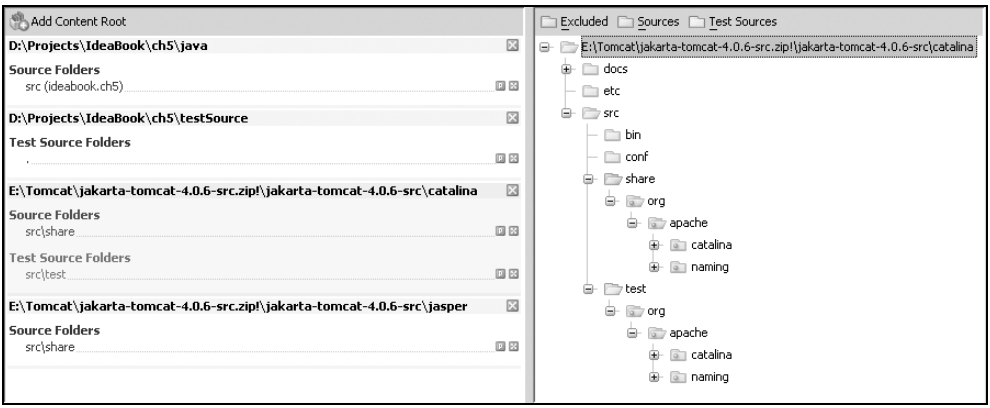


Рис. 5.38. Директории из архива в качестве корневых директорий

5.3.2. Закладка *Libraries (Библиотеки)*

Общий вид закладки с настройками библиотек модуля представлен на рис. 5.39.

На данной закладке собраны опции по настройке библиотек модуля, а также персонального JDK-модуля. Для модуля можно использовать JDK проекта (**Use project JDK**), а можно использовать персональный JDK (**Use module-specific JDK**). JDK можно выбрать из списка, нажав кнопку **OK** (рис. 5.40).

Если необходимого JDK нет в списке доступных, его можно настроить с помощью диалога настройки JDK, который откроется, если нажать кнопку **Edit** (Редактировать). Подробнее о настройке JDK написано в *разделе 4.1.8.1.*

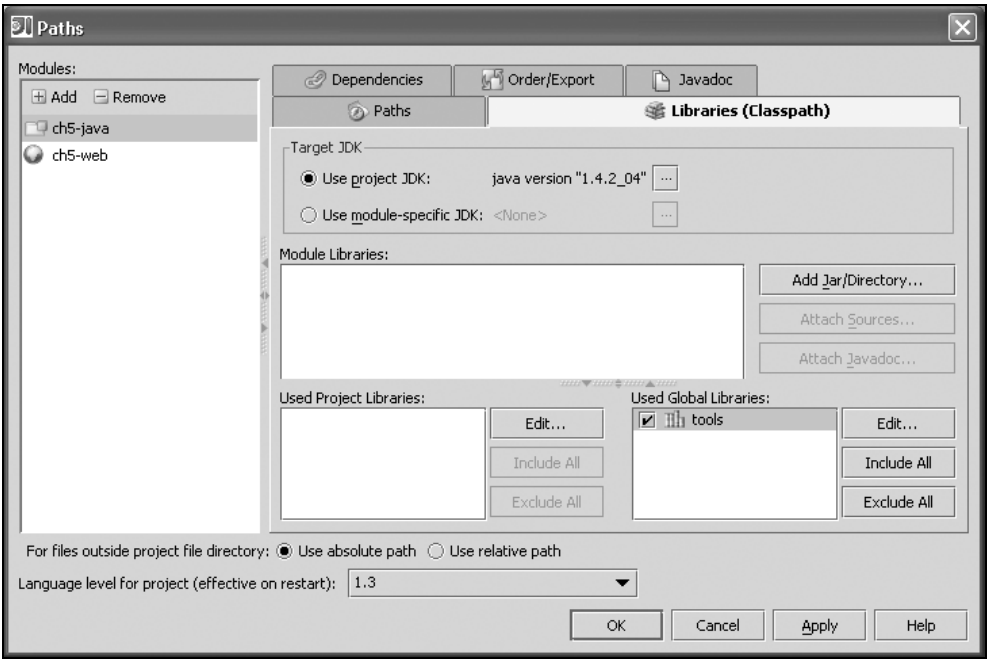


Рис. 5.39. Закладка настройки библиотек модуля

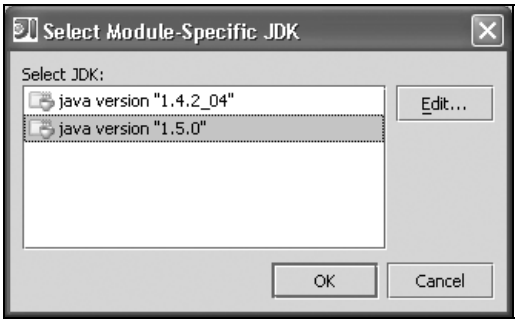


Рис. 5.40. Выбор JDK модуля

Существует три типа библиотек:

- глобальные библиотеки (Global Libraries) — библиотеки, которые можно использовать с любым проектом в IntelliJ IDEA. Подробнее о глобальных библиотеках рассказано в *разделе 4.1.8.2*;
- библиотеки проекта (Project Libraries) — библиотеки, которые могут быть использованы любым модулем в проекте;
- библиотеки модуля (Module Libraries) — библиотеки, актуальные только для конкретного модуля.

Библиотеки модуля, проекта и глобальные библиотеки настраиваются идентично, достаточно сравнить рис. 4.34, 5.41, 5.42.

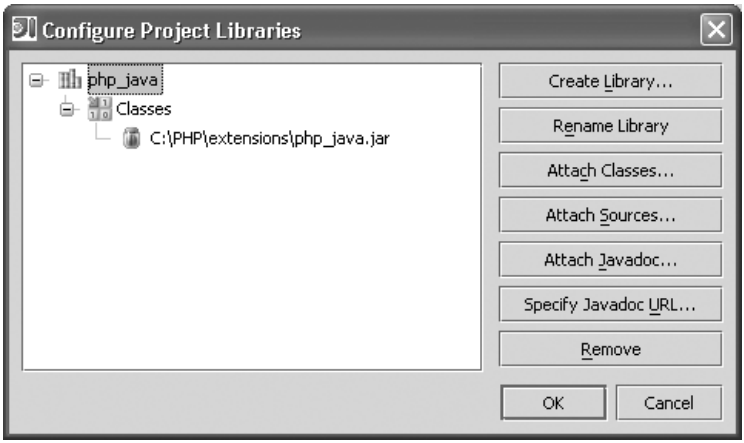


Рис. 5.41. Настройка библиотек проекта

Рис. 5.42. Настройка библиотек модуля

Библиотеки проекта и глобальные библиотеки можно подключить к модулю, поставив галочку напротив необходимой библиотеки в списке. Также можно использовать кнопки **Include All/Exclude All** (Включить все/Исключить все).

Библиотеки модуля также состоят из JAR- или ZIP-архивов, для которых можно настроить комментарии JavaDoc (с помощью кнопки **Attach Javadoc** (Подключить JavaDoc) и **Specify Javadoc URL** (Установить JavaDoc URL)) и исходные тексты (с помощью кнопки **Attach Sources** (Подключить источники)).

5.3.3. Закладка *Dependencies (Зависимости)*

С помощью закладки **Dependencies** (Зависимости) можно задать зависимости между модулями (рис. 5.43).

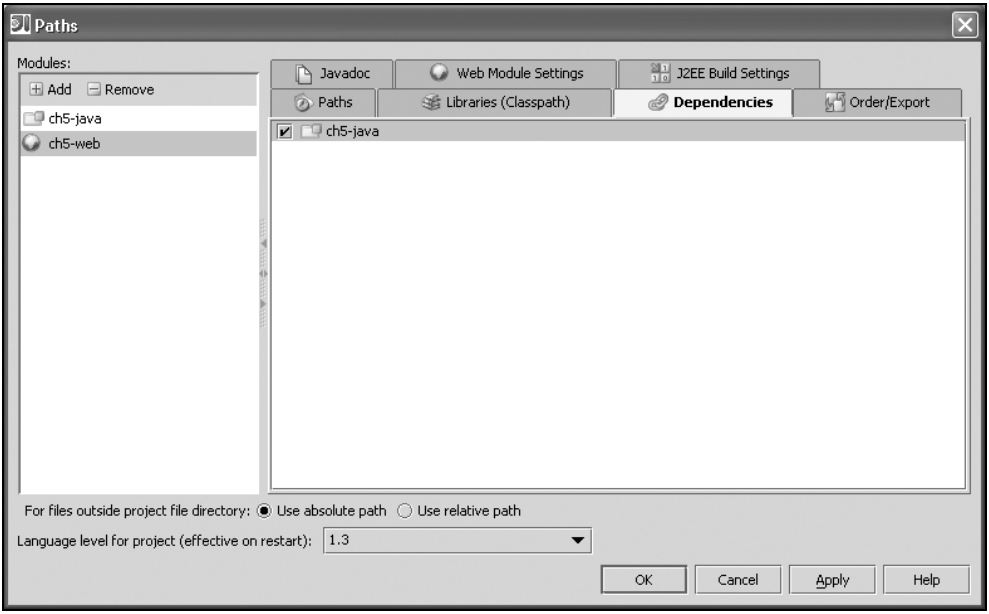


Рис. 5.43. Настройка зависимостей между модулями

Зависимость между модулями устанавливается с помощью галочки, расположенной напротив имени соответствующего модуля.

Зависимость между модулями нужна для того, чтобы определить, файлы какого модуля зависят от файлов другого. Если выполняется компиляция модуля, у которого есть зависимости от других модулей, то первоначально будут скомпилированы зависимые модули.

Модуль, имеющий зависимости от других модулей, использует их выходную директорию в качестве дополнительного пути для поиска файлов (Classpath).

Версия IntelliJ IDEA 4.5 не поддерживает автоматическое распознавание циклических зависимостей между модулями, поэтому за отсутствием такой зависимости приходится следить самостоятельно. В новой версии IntelliJ IDEA 5 предусмотрена автоматическая проверка на циклические зависимости.

5.3.4. Закладка *Order/Export (Порядок/Экспорт)*

Общий вид закладки настроек **Order/Export** (Порядок/Экспорт) представлен на рис. 5.44.

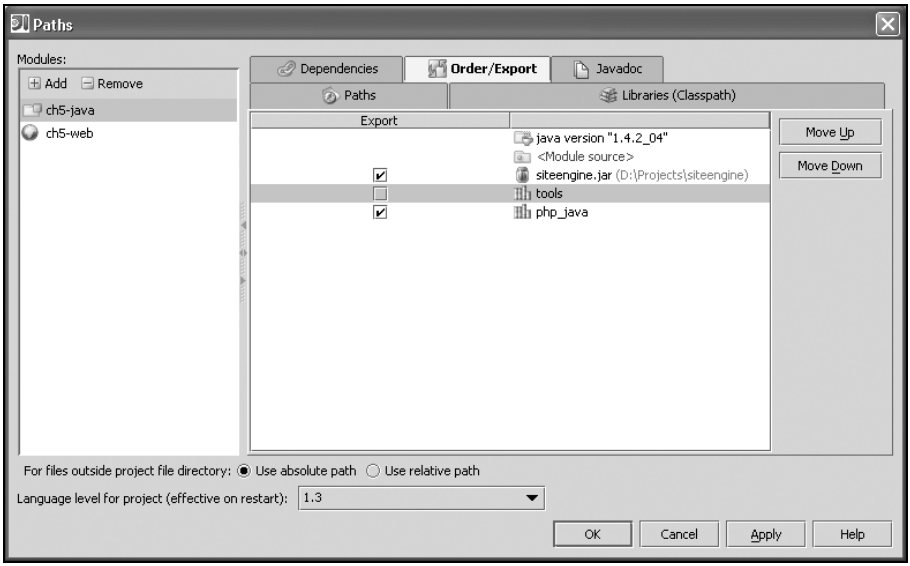


Рис. 5.44. Настройка порядка и экспорта

5.3.4.1. *Order (Порядок)*

С помощью данной закладки можно установить порядок библиотек, в котором они будут использоваться модулем при поиске классов. В большинстве случаев такой порядок не очень важен, однако бывает необходимо четко его специфицировать.

В качестве такого примера можно привести подмену стандартных Java-классов собственными классами. Так как по умолчанию JVM сначала загружает классы из стандартных библиотек, такое переопределение будет невозможным и может привести к ошибке во время исполнения программы. Чтобы этого избежать, можно поставить библиотеку с собственными подменяющими классами перед стандартной библиотекой.

Кстати, при настройке поддержки JSR-14 используется именно такая подмена библиотеки коллекций для модулей (*см. раздел 3.2.11.6*) (рис. 5.45).

Рис. 5.45. Библиотека коллекций установлена первой в списке

Устанавливать порядок использования библиотек можно с помощью кнопок **Move Up/Down** (Переместить вверх/вниз).

5.3.4.2. *Export (Экспорт)*

Экспорт библиотек модуля необходим для того, чтобы другие модули, зависящие от данного модуля, могли использовать и его библиотеки, если это требуется.

Например, в модуле может находиться имплементация некоторых интерфейсов из библиотеки. Чтобы другие модули, которые используют эту имплементацию, могли не включать сам файл библиотеки в список своих библиотек, можно установить экспорт для данной "экспортной" библиотеки модуля.

Библиотека включается в экспорт с помощью установки галочки напротив соответствующего файла в списке (см. рис. 5.45).

5.3.5. Закладка *Javadoc*

С помощью закладки **Javadoc** можно установить файлы (с помощью кнопки **Add Path** (Установить путь)) с комментариями JavaDoc для классов данного модуля (рис. 5.46).

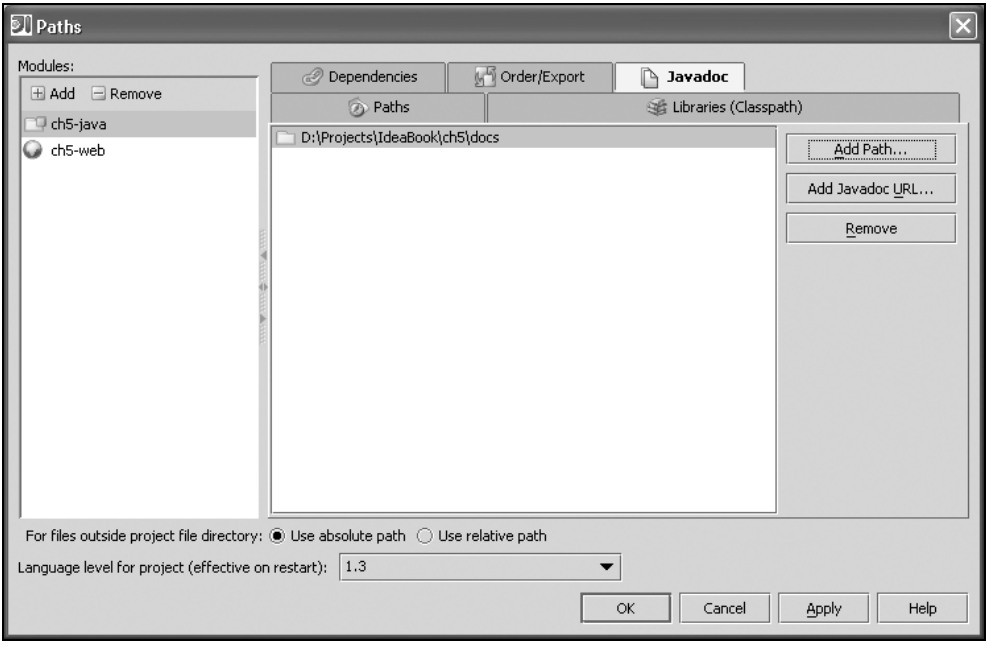


Рис. 5.46. Установка путей с файлами комментариев JavaDoc

Также можно задать URL (с помощью кнопки **Add Javadoc URL**), по которому можно найти необходимые комментарии JavaDoc.

5.3.6. Закладка *Web Module Settings (Свойства Web-модуля)*

Каждый Web-модуль дополнительно к закладкам, описанным в *разделах 5.3.1—5.3.5*, имеет закладку с настройками, представленными на рис. 5.47.

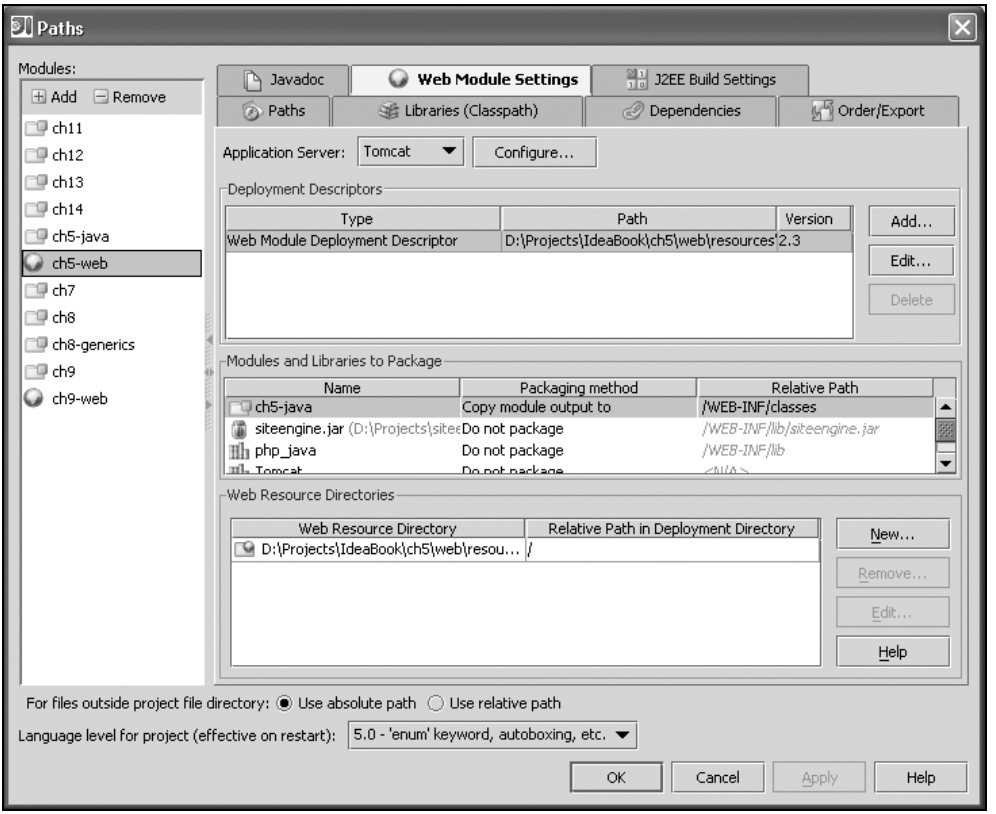


Рис. 5.47. Настройки свойств Web-модуля

На данной закладке расположены следующие настройки:

- **Application Server** (Сервер приложений) — имеется возможность выбрать сервер приложений для использования вместе с данным Web-модулем, а также сконфигурировать набор серверов приложений с помощью кнопки **Configure** (Конфигурировать), которая открывает диалог настройки серверов приложений (*см. раздел 4.1.20*);

- **Deployment Descriptors** (XML-дескриптор Web-модуля) — список доступных XML-дескрипторов для данного Web-модуля. В этом списке содержится тот дескриптор, который мы конфигурировали в *разделе 5.2.3.2*. Дескрипторы можно добавлять (кнопка **Add**), редактировать (кнопка **Edit**) и удалять (кнопка **Delete**). Редактирование дескрипторов происходит с помощью диалога, представленного на рис. 5.12;
- **Modules and Libraries to Package** (Модули и библиотеки для включения в пакет) — перечень модулей и библиотек, которые должны поставляться вместе с Web-модулем. Список модулей и библиотек представлен в виде таблицы с тремя колонками:
  - **Name** (Имя) — имя модуля или библиотеки;
  - **Packaging method** (Метод включения в пакет поставки Web-модуля):
  - **Do not package** (Не включать в пакет) — данный вариант включения доступен и для модулей, и для библиотек;
  - **Copy module output to** (Копировать выходные файлы модуля в пакет) — данный вариант доступен только для модулей;
  - **JAR module output and copy files to** (Архивировать в JAR выходные файлы модуля и включать в пакет) — данный вариант доступен только для модуля;
  - **JAR module, link via manifest and copy to** (Архивировать в JAR выходные файлы модуля, включить ссылку на JAR в файл манифеста и включить в пакет) — данный вариант доступен только для модуля;
  - **Copy files to** (Включить файлы в пакет) — включить файлы библиотеки в пакет. Данный вариант доступен только для библиотеки;
  - **Link via manifest and copy files to** (Включить ссылку на файлы библиотеки в файл манифеста и включить в пакет) — данный вариант доступен только для модуля;
  - **Relative Path** (Относительный путь) — относительный путь, где будут помещены файлы модулей или библиотек при включении в пакет;
- **Web Resource Directories** (Директории с ресурсами Web-модуля). Директории можно добавлять (кнопка **New**), удалять (кнопка **Remove**) и редактировать (кнопка **Edit**). Добавление и редактирование директорий с ресурсами происходит с помощью диалога, представленного на рис. 5.48.

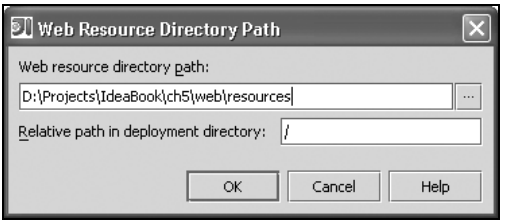


Рис. 5.48. Редактирование директории с ресурсами Web-модуля

5.3.7. Закладка *J2EE Build Settings (Настройка сборки J2EE)*

Все J2EE-модули (Web-модули, Ejb-модули, модули J2EE-приложения) имеют закладку с настройками сборки J2EE. В зависимости от типа модуля на закладке расположены различные опции.

5.3.7.1. Настройка сборки J2EE для Web-модуля

Закладка настройки сборки J2EE для Web-модуля представлена на рис. 5.49.

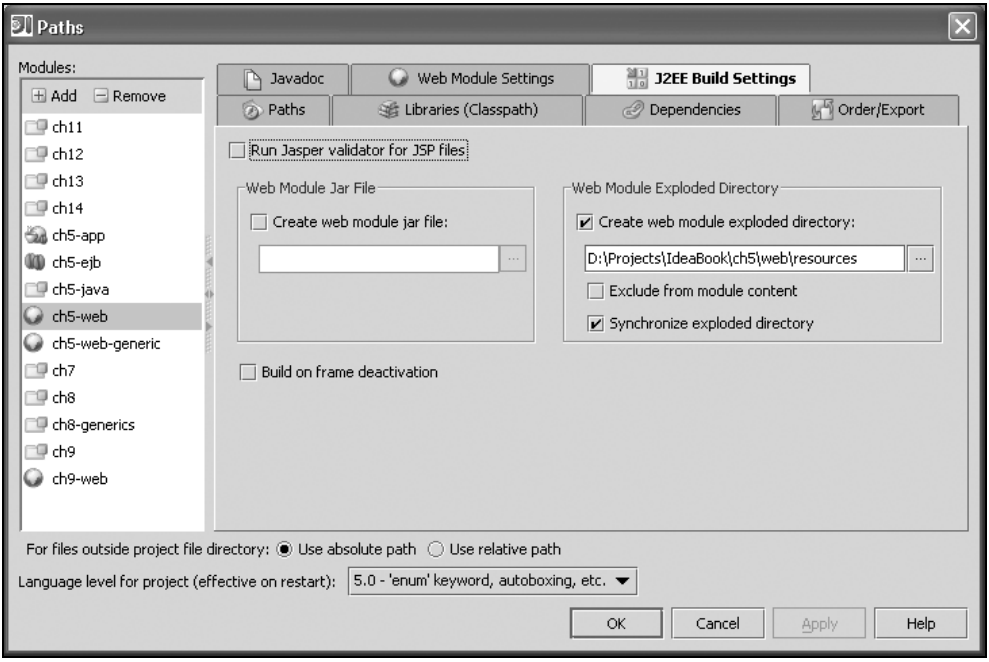


Рис. 5.49. Настройка сборки J2EE для Web-модуля

На данной закладке представлены следующие опции:

- **Run Jasper validator for JSP files** (Запускать утилиту Jasper для проверки JSP-файлов) — данная опция включает автоматическую проверку JSP-файлов при компиляции или сборке проекта;
- **Web Module Jar File** (Файл архива Web-модуля) — опции настройки WAR-архива (Web-Archive) для файлов Web-модуля:
  - **Create web module jar file** (Создавать файл архива для Web-модуля);
  - путь к файлу с архивом;

- **Web Module Exploded Directory** (Выходная директория Web-модуля) — настройки директории Web-модуля, в которую складываются файлы Web-модуля перед запуском Web-приложения:
  - **Create web module exploded directory** (Создавать указанную выходную директорию для Web-модуля);
  - **Exclude from module content** (Исключать выходную директорию из просмотра файлов Web-модуля);
  - **Synchronize exploded directory** (Синхронизировать выходную директорию);
- **Build on frame deactivation** (Выполнять сборку, когда окно IntelliJ IDEA неактивно).

5.3.7.2. Настройка сборки J2EE для Ejb-модуля

Закладка настройки J2EE сборки Ejb-модуля представлена на рис. 5.50.

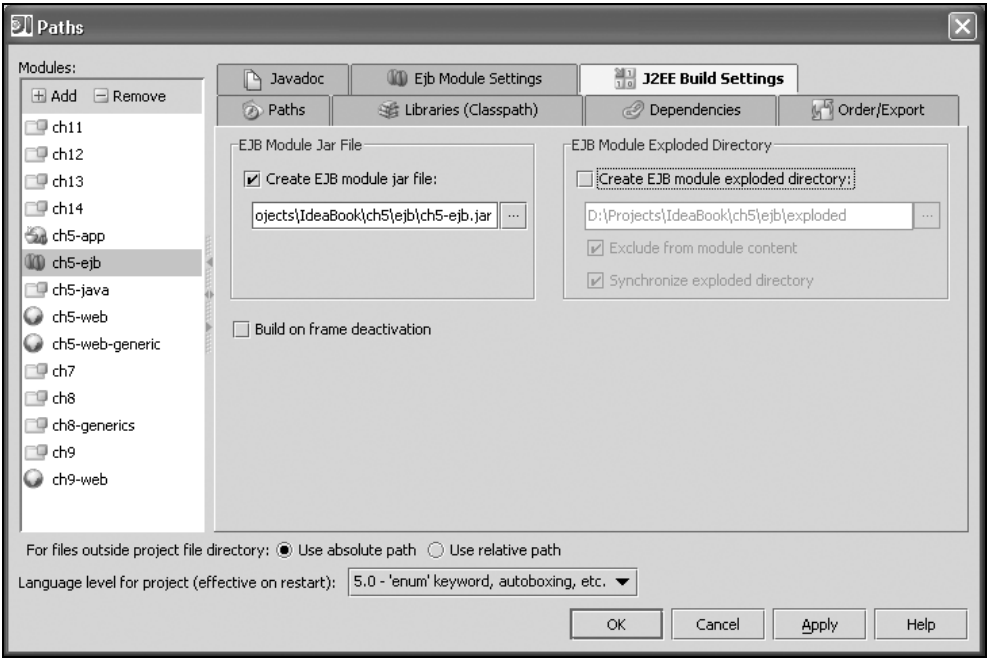


Рис. 5.50. Настройки J2EE сборки для Ejb-модуля

На данной закладке представлены следующие опции:

- **EJB Module Jar File** (JAR-файл для Ejb-модуля):
  - **Create EJB module jar file** (Создавать JAR-файл для EJB-модуля);
  - путь к JAR-файлу;

- **EJB Module Exploded Directory** (Выходная директория EJB-модуля):
  - **Create EJB module exploded directory** (Создавать указанную выходную директорию EJB-модуля);
  - **Exclude from module content** (Исключать выходную директорию из просмотра файлов Web-модуля);
  - **Synchronize exploded directory** (Синхронизировать выходную директорию);
- **Build on frame deactivation** (Выполнять сборку, когда окно IntelliJ IDEA неактивно).

5.3.7.3. Настройка сборки J2EE для модуля J2EE-приложения

Закладка настройки J2EE-сборки для модуля J2EE-приложения представлена на рис. 5.51.

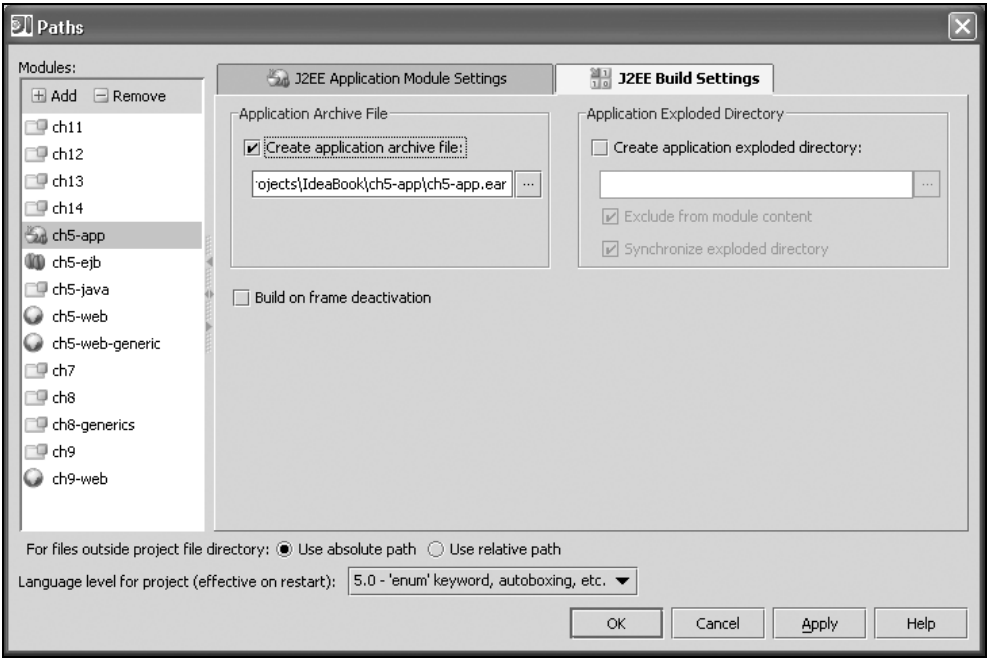


Рис. 5.51. Настройка J2EE-сборки для модуля J2EE-приложения

На данной закладке представлены следующие опции:

- **Application Archive File** (Файл архива приложения):
  - **Create application archive file** (Создавать файл архива для приложения);
  - путь к файлу архива;

- **Application Exploded Directory** (Выходная директория приложения):
  - **Create application exploded directory** (Создавать указанную выходную директорию приложения);
  - **Exclude from module content** (Исключать выходную директорию из просмотра файлов Web-модуля);
  - **Synchronize exploded directory** (Синхронизировать выходную директорию);
- **Build on frame deactivation** (Выполнять сборку, когда окно IntelliJ IDEA неактивно).

5.3.8. Закладка *EJB Module Settings (Свойства EJB-модуля)*

Каждый EJB-модуль, кроме закладок **Paths** (Пути), **Libraries** (Библиотеки), **Dependencies** (Зависимости), **Order/Export** (Порядок/Экспорт), **Javadoc**, имеет две дополнительные закладки: **Ejb Module Settings** (Свойства Ejb-модуля) и **J2EE Build Settings** (Настройки сборки J2EE).

Закладка **Ejb Module Settings** (Свойства Ejb-модуля) представлена на рис. 5.52. Настройки сборки J2EE для Ejb-модуля описаны в *разделе 5.3.7.2.*

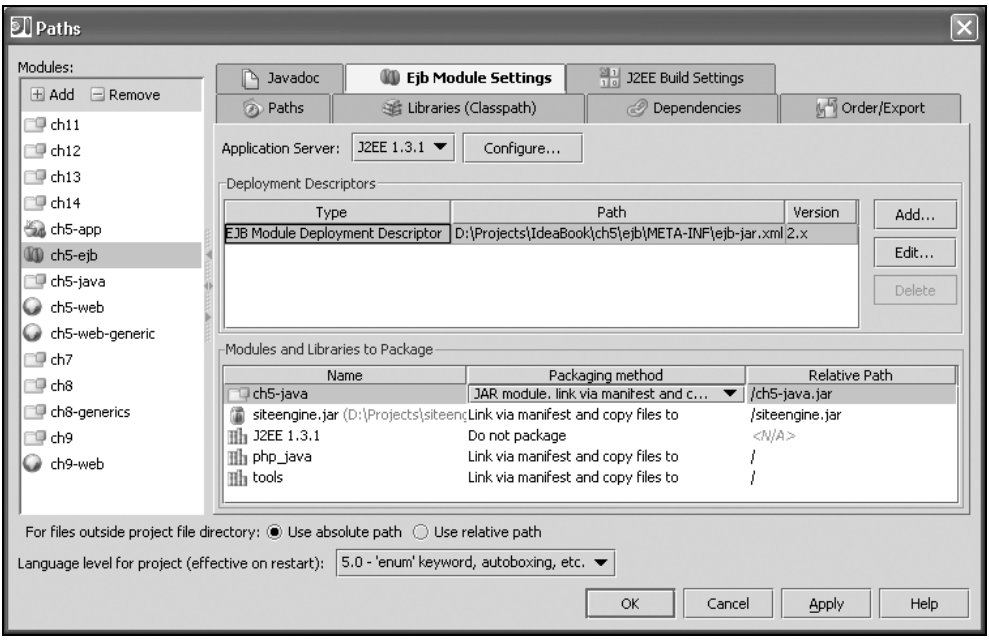


Рис. 5.52. Настройки Ejb-модуля

На данной закладке представлены следующие опции и параметры:

- **Application Server** (Сервер приложений) — имеется возможность выбрать сервер приложений для использования вместе с данным Ejb-модулем, а также сконфигурировать набор серверов приложений с помощью кнопки **Configure** (Конфигурировать), которая открывает диалог настройки серверов приложений (*см. раздел 4.1.20*);
- **Deployment Descriptors** (XML-дескриптор EJB-модуля) — список доступных XML-дескрипторов для данного EJB-модуля. В этом списке содержится тот дескриптор, который мы конфигурировали в *разделе 5.2.3.3*. Дескрипторы можно добавлять (кнопка **Add**), редактировать (кнопка **Edit**) и удалять (кнопка **Delete**). Редактирование дескрипторов происходит с помощью диалога, представленного на рис. 5.25;
- **Modules and Libraries to Package** (Модули и библиотеки для включения в пакет) — перечень модулей и библиотек, которые должны поставляться вместе с EJB-модулем. Список модулей и библиотек представлен в виде таблицы с тремя колонками:
  - **Name** (Имя) — имя модуля или библиотеки;
  - **Packaging method** (Метод включения в пакет поставки Ejb-модуля):
  - **Do not package** (Не включать в пакет) — данный вариант включения доступен и для модулей, и для библиотек;
  - **Copy module output to** (Копировать выходные файлы модуля в пакет) — данный вариант доступен только для модулей;
  - **JAR module, link via manifest and copy to** (Архивировать в JAR выходные файлы модуля, включить ссылку на JAR в файл манифеста и включить в пакет) — данный вариант доступен только для модуля;
  - **Link via manifest and copy files to** (Включить ссылку на файлы библиотеки в файл манифеста и включить в пакет) — данный вариант доступен только для модуля;
  - **Relative Path** (Относительный путь) — относительный путь, где будут помещены файлы модулей или библиотек при включении в пакет.

5.3.9. *J2EE Application Module Settings (Свойства модуля J2EE приложения)*

Модули J2EE-приложения имеют всего две закладки с настройками: **J2EE Application Module Settings** (Свойства модуля J2EE-приложения) и **J2EE Build Settings** (Настройки сборки J2EE). Закладка с настройками свойств модуля J2EE приложения представлена на рис. 5.53. Настройка параметров J2EE сборки для J2EE-приложения описана в *разделе 5.3.7.3*.

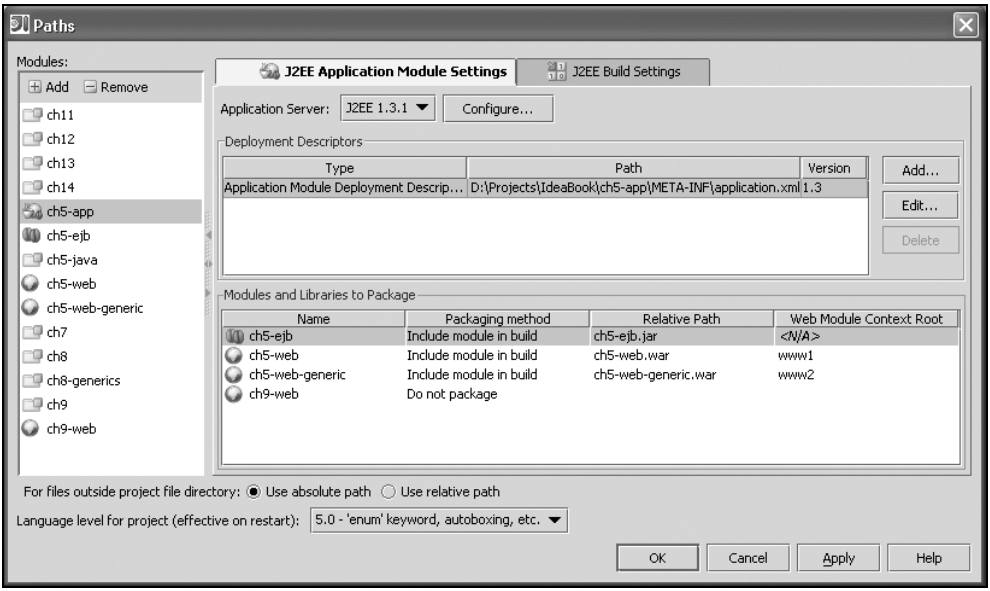


Рис. 5.53. Настройка модуля J2EE-приложения

На данной закладке представлены следующие опции и параметры:

- **Application Server** (Сервер приложений) — имеется возможность выбрать сервер приложений для использования вместе с данным модулем J2EE-приложения, а также сконфигурировать набор серверов приложений с помощью кнопки **Configure** (Конфигурировать), которая открывает диалог настройки серверов приложений (*см. раздел 4.1.20*);
- **Deployment Descriptors** (XML-дескриптор модуля J2EE-приложения) — список доступных XML-дескрипторов для данного модуля J2EE-приложения. В этом списке содержится тот дескриптор, который мы конфигурировали в *разделе 5.2.3.4*. Дескрипторы можно добавлять (кнопка **Add**), редактировать (кнопка **Edit**) и удалять (кнопка **Delete**). Редактирование дескрипторов происходит с помощью диалога, представленного на рис. 5.30;
- **Modules and Libraries to Package** (Модули и библиотеки для включения в пакет) — перечень модулей, которые должны поставляться вместе с модулем J2EE-приложения. Список модулей представлен в виде таблицы с несколькими колонками:
  - **Name** (Имя) — имя модуля или библиотеки;
  - **Packaging method** (Метод включения в пакет поставки EJB-модуля):
  - **Do not package** (Не включать в пакет);
  - **Include module in build** (Включить модуль в сборку);

- **Relative Path** (Относительный путь) — относительный путь, где будут помещены файлы модулей при включении в пакет;
- **Web Module Context Root** (Путь контекста Web-модуля) — в данной колонке можно задать путь контекста Web-модуля относительно J2EE-приложения.

Примечание

Несмотря на то, что таблица со списком модулей озаглавлена как **Modules and Libraries to Package** (Модули и библиотеки для включения в пакет), на самом деле в данной таблице доступны только модули, что очевидно. Название несколько неточное.

5.4. Создание проекта из имеющихся файлов

Рассмотрим вариант создания проекта, когда у нас уже имеется некоторый набор Java-файлов, для которых мы хотим сделать проект в IntelliJ IDEA.

В качестве примера рассмотрим создание проекта для классов из примеров, поставляемых совместно с JDK в каталоге demo.

Для создания такого проекта нужно использовать мастер создания проекта и выполнить почти все шаги, которые мы рассмотрели в *разделе 5.2.1*.

Последовательно введем информацию о каталоге проекта (рис. 5.54).

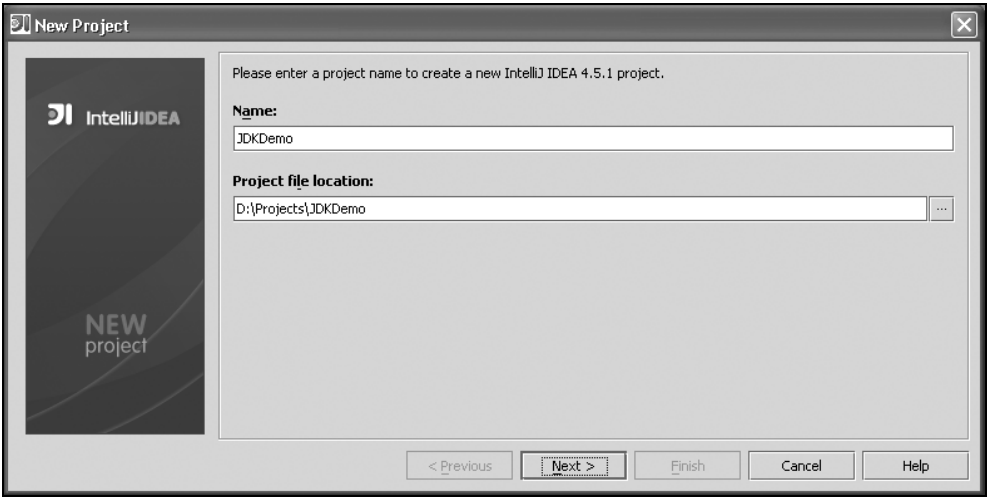


Рис. 5.54. Выбор директории для проекта

Мы создадим модуль для демонстрационных классов FileChooserDemo из JDK/demo (рис. 5.55).

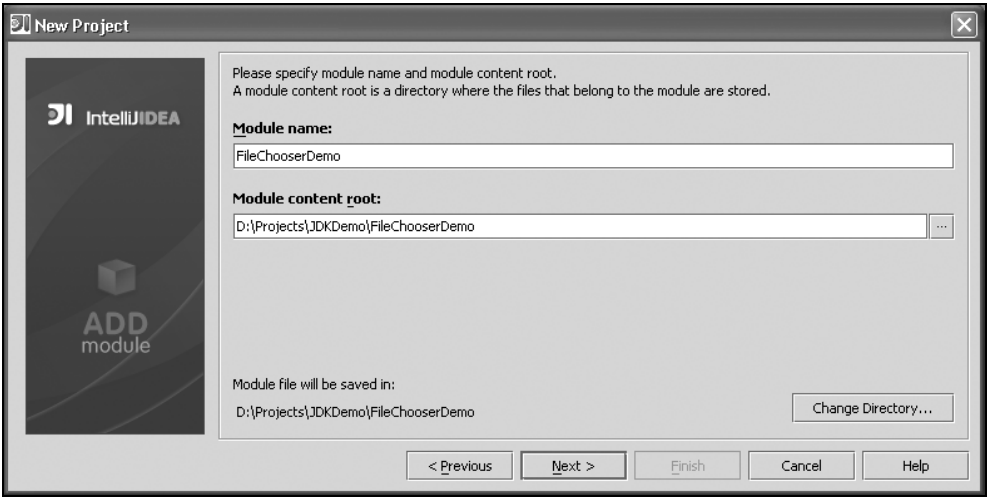


Рис. 5.55. Выбор места расположения файла модуля

На следующем шаге мы не будет создавать директорию для исходных файлов (рис. 5.56), а добавим ее позже.

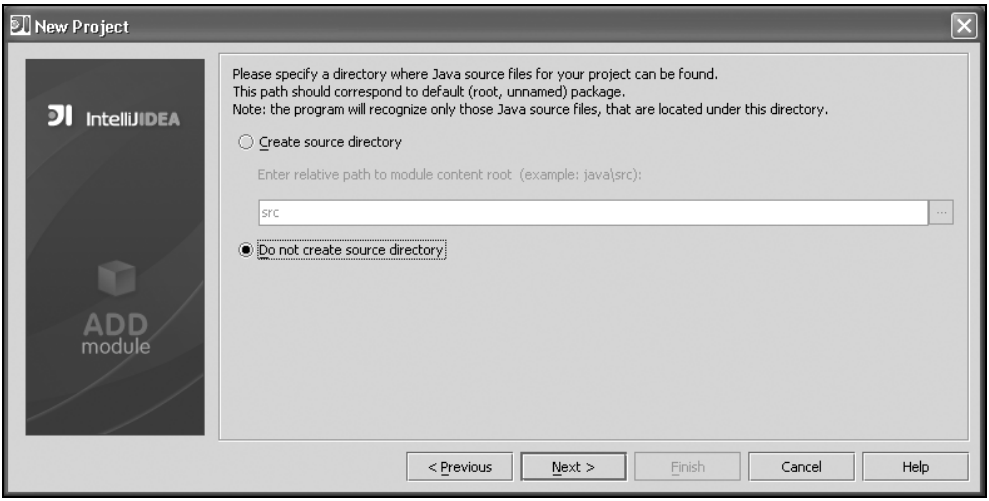


Рис. 5.56. Директория для исходных файлов не создается

В итоге у нас получился проект с пустым модулем (рис. 5.57).

Теперь мы откроем диалог настройки модуля и добавим в него корневую директорию с файлом FileChooserDemo (рис. 5.58).

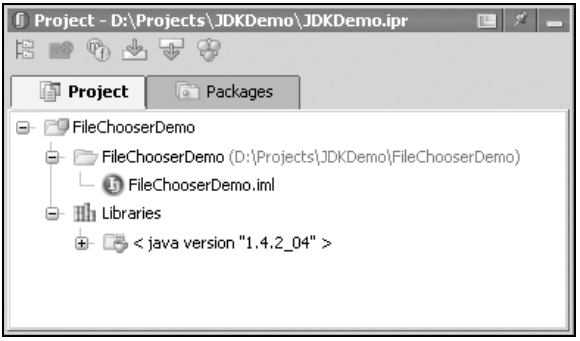


Рис. 5.57. Проект с пустым модулем

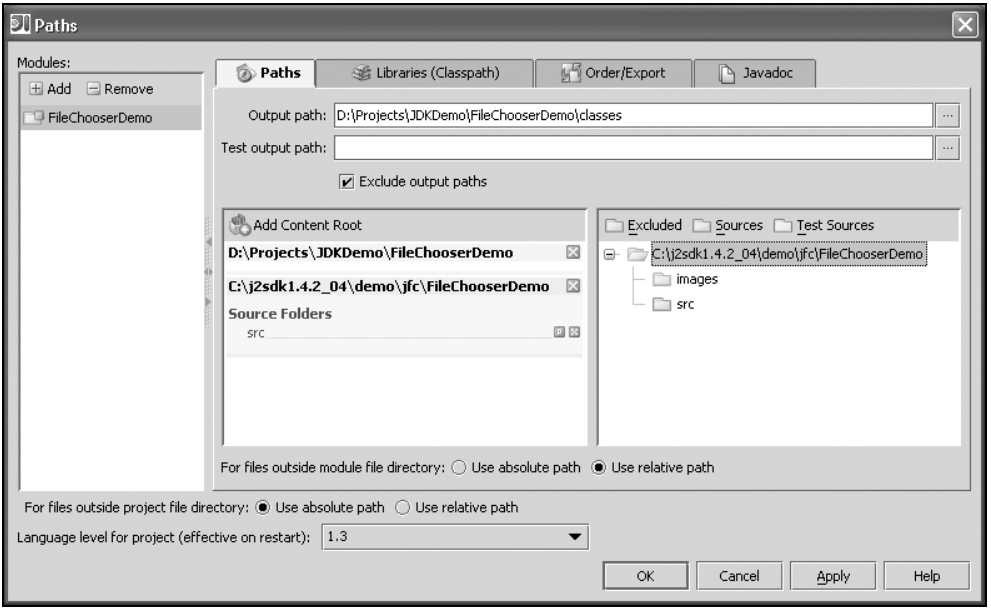


Рис. 5.58. Корневая директория с файлом FileChooserDemo добавлена

IntelliJ IDEA автоматически определила, что вложенная директория src содержит исходные файлы, и пометила ее соответствующим образом.

В итоге наш проект примет вид, представленный на рис. 5.59.

Если мы попробуем скомпилировать и запустить файл FileChooserDemo, все отлично запустится.

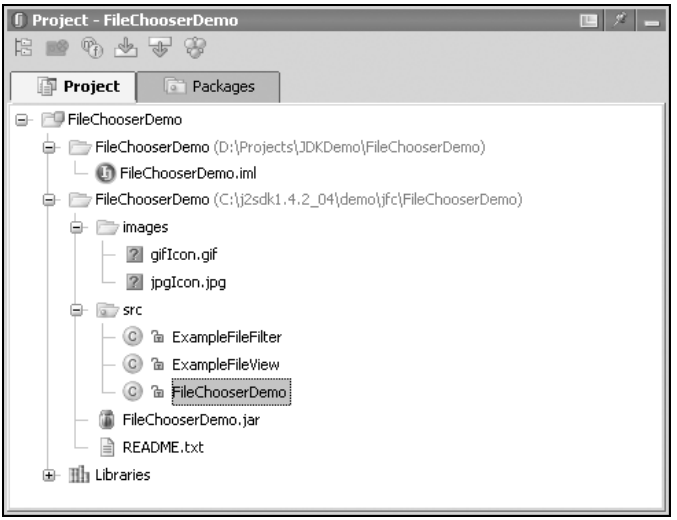


Рис. 5.59. Проект после добавления дополнительной корневой директории

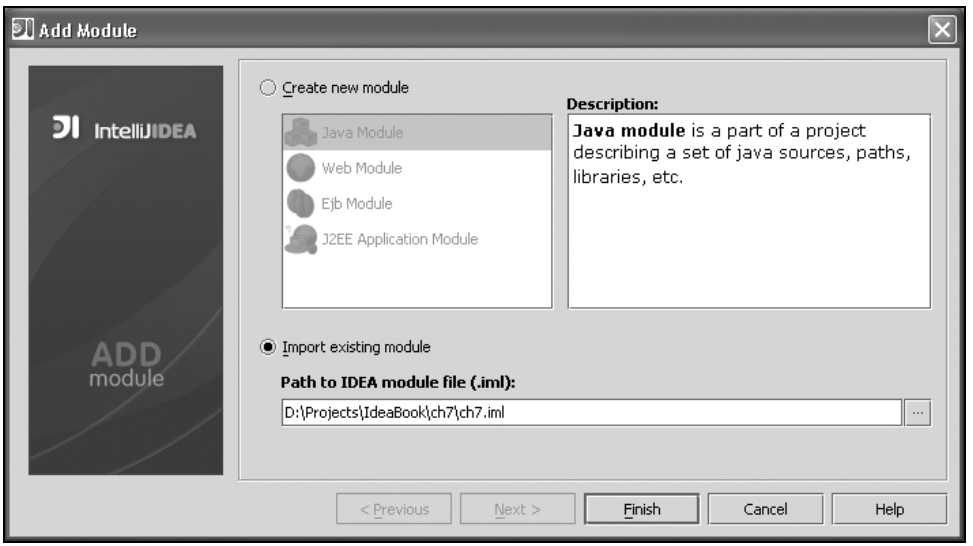
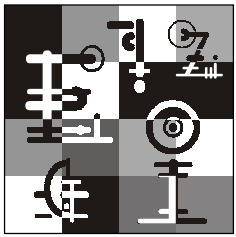


Рис. 5.60. Добавление существующего модуля

Аналогично мы можем добавить модули для всех демонстрационных примеров из JDK.

Примечание

Если в проект нужно добавить уже существующий модуль из другого проекта IntelliJ IDEA, надо выполнить команду **File | New Module** (Файл | Новый модуль) (*см. раздел 3.2.1.2*) и в диалоге указать путь к существующему модулю (рис. 5.60).



Глава 6

Работа с кодом

IntelliJ IDEA обладает замечательным, с точки зрения программиста, редактором. Мы уже рассказывали о редакторе IntelliJ IDEA в *разделе 3.6*. Сейчас же мы хотим подробнее рассмотреть сам процесс редактирования и создания кода.

Прежде, программируя на Java или других языках, мы использовали либо специальные среды разработки вроде Microsoft Visual Studio, Visual Slick Edit, либо просто хорошие редакторы с некоторыми дополнительными возможностями вроде встроенного редактора FAR или Vi. Максимум, что можно было ждать от редактора, это подсветки ключевых слов языка, может быть, подсветки неправильной конструкции кода. Было здорово, если работала какая-либо вспомогательная система завершения кода, вроде выбора метода или поля, после разделителя класса.

В IntelliJ IDEA таких функций, помогающих в написании кода, намного больше. Сейчас, проработав с IntelliJ IDEA уже более двух лет, мы даже не замечаем, насколько привыкли к возможностям ее редактора.

Недавно мне пришлось писать небольшой скрипт на PHP. Я писал его во встроенном редакторе FAR. Каждые несколько секунд я буквально утыкался носом в то, что пытался вызвать функции автоматического завершения кода, просмотреть информацию о параметрах, использовать "живой шаблон", которых в FAR, естественно, не было. В итоге код в сотню строк на PHP я писал раза в три дольше, чем мог бы написать такой же код на Java. И это не потому, что PHP настолько плохой язык. Все дело в редакторе.

Использование IntelliJ IDEA делает программиста зависимым от возможностей этой среды настолько, что мы до сих пор не знаем ни одного человека, который, проработав некоторое время на IntelliJ IDEA, перешел бы на другой редактор или IDE.

К комфорту слишком быстро привыкаешь ☺.

В этой главе мы расскажем о создании и редактировании кода, его анализе на предмет возможных ошибок, повторов и зависимостей. В *разделе 6.5* мы рассмотрим структурный поиск и замену кода.

6.1. Работа с редактором

Мы выделили следующие аспекты создания кода в редакторе, которые мы считаем ключевыми:

- завершение кода (completion);
- подсветка;
- "подсказки" (intention actions);
- форматирование кода;
- дополнительная информация о коде;
- работа с буфером обмена;
- сокращенное представление кода.

Каждому аспекту мы посвятили отдельный раздел. Мы подумали, что самым удобным и понятным способом рассказать о создании кода с помощью редактора — это показать на практических примерах. Именно так мы и поступим. Вначале будем давать краткую информацию о той или иной функции, а затем показывать, как ее можно использовать на реальном примере.

Предположим, что у вас уже есть настроенный проект в IntelliJ IDEA (если еще нет, *см. главу 5*), в котором пока нет ни единого класса.

С этого мы и начнем — создадим класс. Удобно, когда каждому Java-классу или интерфейсу соответствует отдельный файл (за исключением внутренних, конечно).

Для создания класса в IntelliJ IDEA нужно выполнить команду **Edit | New** (Редактирование | Создать) (<Alt>+<Insert>) для той директории, в которой вы хотите что-то создать. Директория может быть открыта в инструментальном окне **Project** (Проект) или **Commander** (Коммандер) (*см. раздел 3.4.8*). Мы вызвали эту команду из инструментального окна **Project** (Проект) (рис. 6.1).

В представленном контекстном меню указаны все возможные сущности, которые можно создать в данной директории. Подробнее об этих вариантах можно прочитать в *разделе 6.3.1*.

Мы хотим создать класс, поэтому выбираем первый пункт и в открывшемся диалоге вводим имя нашего класса — `Book`. Класс `Book`, а вместе с ним и файл Book.java, в котором он записан, готов (листинг 6.1).

Листинг 6.1

```
public class Book {
}
```

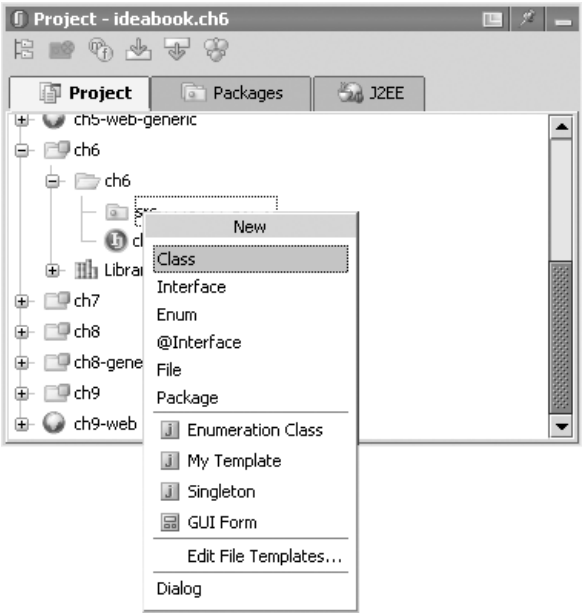


Рис. 6.1. Вызов команды **Edit** | **New**

Вот такой простой класс. Тем не менее, если подсчитать, сколько клавиш мы для этого нажали — семь, если быть точными, окажется, что символов в описании этого класса почти в три раза больше. IntelliJ IDEA сэкономила треть нашего времени. И это только начало!

Обычно, после того как класс создан, в него добавляется некоторая начинка, например поля и методы.

Для примера, не долго думая, мы решили создать структуру классов, отражающих структуру книги с точки зрения писателя. У книги непременно должен быть хотя бы один автор и название. Добавим в наш класс соответствующие поля и методы.

Мы начали писать название модификатора доступа `private` и уже написали букву '`p`', после чего вызвали команду **Code | Complete Code | Basic** (Код | Завершить код | Простое завершение) (<Ctrl>+<Space>), и вот, IntelliJ IDEA уже предлагает нам на выбор три варианта, первый из которых мы и выбираем (рис. 6.2) (подробнее о команде простого завершения кода написано в *разделе 6.1.1*).

Затем, мы начали писать тип поля класса `String` и написали первые три буквы, после чего снова выполнили команду простого завершения кода, и IntelliJ IDEA снова предложила нам варианты завершения, один из которых вполне нас устроил (рис. 6.3).

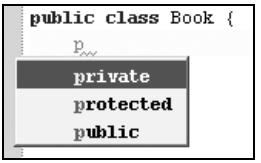


Рис. 6.2. Простое завершение кода для ключевого слова

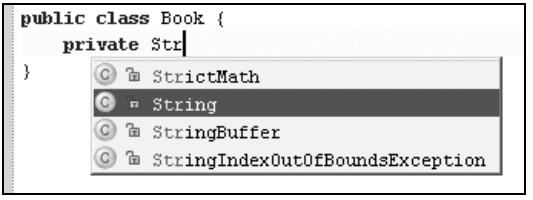


Рис. 6.3. Простое завершение кода для типа из текущей области видимости

После чего мы уже сами записали название поля.

Аналогичным образом мы поступили и со вторым полем, в результате чего у нас получился класс, представленный в листинге 6.2 (рис. 6.4).

Листинг 6.2

```
public class Book {
    private String title;
    private String author;
}
```

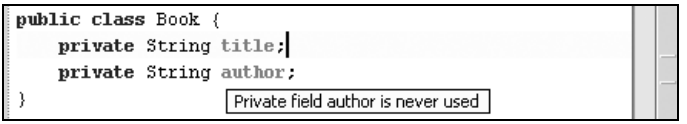


Рис. 6.4. Класс в редакторе

Как можно заметить на рис. 6.4, ключевые слова языка Java написаны жирным шрифтом темно-синего цвета (к сожалению, цвет на черно-белой иллюстрации вы увидеть не можете). Поля класса тоже выделены жирным шрифтом, но серого цвета. IntelliJ IDEA имеет очень мощные возможности подсветки всего, что только можно структурировать в Java-коде. Подробнее о подсветке рассказано в *разделе 6.1.2*.

Серый цвет имен полей класса — сигнал, что с ними что-то не так (по-хорошему они должны быть темно-бордовыми), да и две желтые метки в правом поле редактора свидетельствуют о замечаниях к нашему коду. Если подвести курсор мыши к одному из полей, все станет ясно. IntelliJ IDEA

выведет всплывающее окно с сообщением **Private field author is never used** — поле `author`, имеющее модификатор доступа `private`, не используется — непорядок.

Вроде бы код, с точки зрения языка Java, корректен, однако толку от него — ноль. IntelliJ IDEA это замечает и предупреждает. Мы и сами собирались создавать методы установки и считывания данных полей класса, но с подсказкой лучше — IntelliJ IDEA не даст забыть.

Если подвести курсор к названию поля, слева в строке загорится желтая лампочка, свидетельствующая о том, что у IntelliJ IDEA появилось какое-то предложение (рис. 6.5).

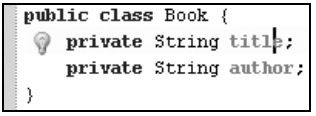


Рис. 6.5. У IntelliJ IDEA появилась какая-то идея

Если подвести к лампочке курсор мыши и нажать на нее, раскроется список "подсказок" — действий, которыми IntelliJ IDEA предлагает исправить часть кода, вызвавшую предупреждение (рис. 6.6).

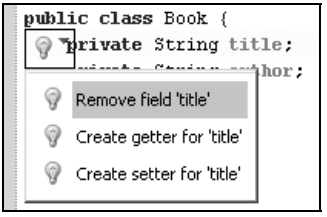


Рис. 6.6. "Подсказки" IntelliJ IDEA

Такая "подсказка" в терминах IntelliJ IDEA называется "Intention Action" — намеренное действие к исправлению проблемы. Мы будем называть это "подсказкой". О "подсказках" мы подробнее расскажем в *разделе 6.1.3*.

Из предложенных в "подсказке" действий, нас вполне устраивают второе и третье, поскольку первый вариант, хоть и исправит положение с неиспользуемой переменной, совсем нам не подходит. Выберем второе действие. IntelliJ IDEA автоматически создаст для нашего поля метод `getTitle()`. Тем не менее серая подсветка с поля не исчезнет, и, подведя курсор к полю, мы увидим почему (рис. 6.7).

В сообщении сказано: "Полю с модификатором `private` никогда не присваивается значение". Что ж, мы полностью с этим согласны. Использовав второй раз "подсказку", содержащую на этот раз лишь одно действие (создать метод `setTitle()` для поля), мы устраним все "претензии к коду". Ана-

логичным образом поступим и со вторым полем. В результате у нас получится класс, код которого представлен в листинге 6.3.

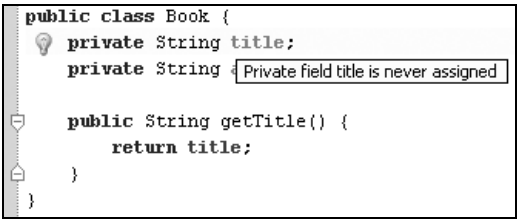


Рис. 6.7. Еще одно замечание к коду

Листинг 6.3

```
public class Book {
    private String title;
    private String author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

В редакторе цвет полей стал темно-бордовым, с правого поля исчезли желтые метки, а индикатор анализа текста стал зеленым (*см. раздел 3.6.2.1*). Замечательно.

Кстати, обратите внимание на выравнивание, которое имеет код в листинге 6.3. IntelliJ IDEA сама выровняла текст, а между методами вставила до-

полнительный перевод строки. Это сработала функция автоматического форматирования кода, о которой мы подробнее расскажем в *разделе 6.1.4*.

Итак, если не считать, что нам пришлось подробно описывать каждое действие, мы потратили на создание кода нашего класса очень мало времени. Тем не менее код его корректен, выровнен и готов к дальнейшему использованию — именно то, что нужно программисту.

Теперь не хватает только конструктора, который мы создадим с помощью команды **Code | Generate** (Код | Создать) (<Alt>+<Insert>), выбрав из предложенного списка пункт **Constructor** (Конструктор) и указав необходимые для инициализации поля (рис. 6.8).

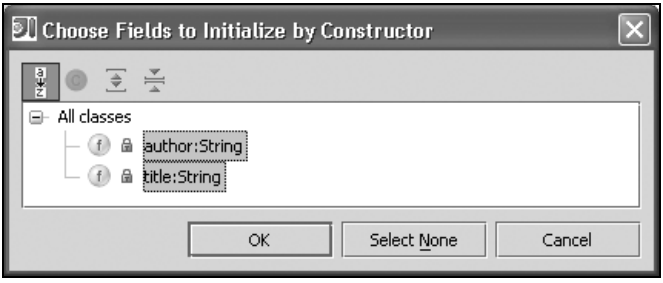


Рис. 6.8. Выбор полей для инициализации конструктором

И конструктор готов. Подробнее об автоматической генерации кода рассказано в *разделе 6.3*.

С нашим кодом все хорошо, за исключением одной вещи — у книги может быть несколько авторов. Надо это учесть, изменив поле `author` типа `String` на список авторов `authors` типа `List<Author>`. Мы хотим использовать пока не существующий класс `Author`, в котором будет содержаться информация об авторах.

Примечание

Мы используем возможности языка Java версии 5, которые полностью поддерживаются IntelliJ IDEA (*см. главу 11*). Однако все, о чем мы будем рассказывать, справедливо и для более старых версий языка.

Внесем соответствующие исправления в код нашего класса (рис. 6.9).

IntelliJ IDEA сразу же подсветила наш код красным — все правильно, ведь старое поле исчезло, также ничего не известно о новом классе `Author`.

Начнем исправлять код с класса `Author`. Если подвести курсор к упоминанию имени этого класса в коде, IntelliJ IDEA выведет "подсказку", но вместо прежней лампочки там будет лампочка с восклицательным знаком (рис. 6.10).

```
public class Book {
    private String title;
    private List<Author> authors;

    public Book(String author, String title) {
        this.author = author;
        this.title = title;
    }
```

Рис. 6.9. Фрагмент измененного кода в редакторе

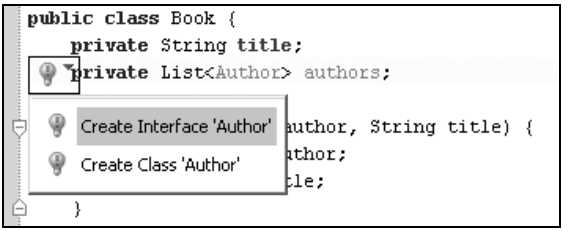


Рис. 6.10. "Подсказка" с лампочкой и восклицательным знаком

Такая подсказка в терминах IntelliJ IDEA называется "Quick Fix" — быстрое исправление. Данный тип "подсказок" появляется тогда, когда с кодом не просто что-то не так, а совсем плохо. Как в нашем случае. Данная подсказка предлагает нам на выбор два действия — создать интерфейс `Author` или создать класс `Author`. Мы выбираем второй.

IntelliJ IDEA попросит нас указать пакет, в котором должен быть создан класс (рис. 6.11) после чего класс будет создан.

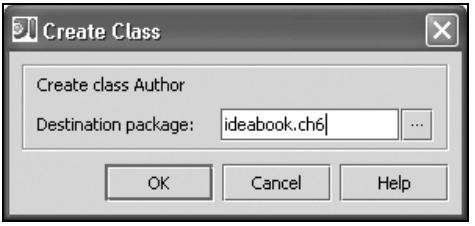


Рис. 6.11. Выбор пакета для создания класса `Author`

Исправив эту ошибку, IntelliJ IDEA примется за следующие. Теперь ей не понравился класс `List`, который мы использовали для указания типа. Среди доступных классов (включая стандартные классы из JDK) имеется несколько классов с таким именем, из которых нам и предложит выбрать IntelliJ IDEA (рис. 6.12).

Нас как раз устроит интерфейс `java.util.List`, который мы и выберем из списка. В ответ на это IntelliJ IDEA добавит в код нашего класса строку

импорта интерфейса `java.util.List`. Именно то, что нам было нужно. Так работает функция автоматического импортирования, о которой мы подробнее расскажем в *разделе 6.3.10*.

Рис. 6.12. Предложение выбрать класс из списка доступных

Теперь нам придется сделать немного ручной работы — удалить старые методы `getAuthor()` и `setAuthor()` для поля `author`. Еще мы удалим конструктор — затем, чтобы создать новый. После чего создадим методы `getAuthors()` и `setAuthors()` для нового поля `authors`, а также создадим новый конструктор, который будет принимать новые параметры. В результате у нас получился класс, представленный в листинге 6.4.

Листинг 6.4

```
import java.util.List;

public class Book {
    private String title;
    private List<Author> authors;

    public Book(List<Author> authors, String title) {
        this.authors = authors;
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public List<Author> getAuthors() {
```

```
        return authors;
    }

    public void setAuthors(List<Author> authors) {
        this.authors = authors;
    }
}
```

Теперь самое время заняться классом `Author`, который был создан для нас IntelliJ IDEA. Есть много способов открыть этот класс в редакторе. Например:

- **Go To | Class** (Перейти | Класс) (<Ctrl>+<N>) — просто выполнить команду в любом месте интерфейса IntelliJ IDEA. Подробнее *см. раздел 6.2.1*;
- **Go To | Declaration** (Перейти | Определение) (<Ctrl>+<B>) — подвести курсор к имени класса в коде и выполнить команду. Можно просто щелкнуть левой кнопкой мыши на имени класса, удерживая нажатой клавишу <Ctrl>. Подробнее *см. раздел 6.2.4*;
- **View | Recent Files** (Просмотр | Последние файлы) (<Ctrl>+<E>) — найти класс в списке последних редактировавшихся классов. Подробнее *см. раздел 6.2.7*;
- **Windows | Select Next/Previous Tab** (Окно | Выбрать следующую/предыдущую закладку редактора) (<Alt>+<Right>/<Left>). Подробнее *см. раздел 6.2.6*;
- Можно придумать еще ☺.

Как вы видите, вариантов множество. В каждом конкретном случае удобно использовать какой-то из способов. Тут все еще зависит от привычек. Можно ведь и просто щелкнуть на имени класса в инструментальном окне **Project** (Проект). Кому что удобнее.

В данный момент мы нажали комбинацию клавиш <Alt>+<Right> и перешли на соседнюю закладку редактора, на которой открыт класс `Author`.

Добавим в него поле `name` типа `java.lang.String` и на этом ограничимся. Ну, конструктор и методы `getName()`/`setName()`, разумеется, тоже сделаем. Вернее IntelliJ IDEA это сделает за нас (листинг 6.5).

Листинг 6.5

```
public class Author {
    private String name;

    public Author(String name) {
```

```
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Теперь поработаем над классами, представляющими структуру книги. Наша книга состоит из частей, те, в свою очередь, состоят из глав, которые состоят из разделов. Отразим эту структуру в классах.

Начнем с класса `Section` (рис. 6.13).

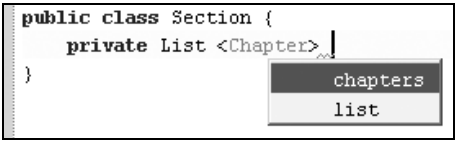


Рис. 6.13. Создание класса `Section`, завершение имени переменной

Дойдя до того места, в котором на рис. 6.13 стоит курсор, мы вызвали команду простого завершения кода (<Ctrl>+<Space>), и IntelliJ IDEA предложила нам два варианта имени переменной, первый из которых мы и выбрали. Когда мы только начинали использовать IntelliJ IDEA, такое "волшебство" было очень удивительным — откуда она знает, как мы хотели назвать это поле? Потом к этому привыкаешь ☺.

Вообще, команда простого завершения кода (<Ctrl>+<Space>) используется очень часто. Так, например, за время написания примеров кода к данной книге, один из авторов использовал команду простого завершения кода 858 раз, что в десять раз больше, чем следующая из команд в списке статистики (рис. 6.14).

Для просмотра статистики мы использовали команду **Help | Productivity Guide** (Помощь | Руководство по продуктивности).

Классы, которые мы создали, представлены в листинге 6.6.

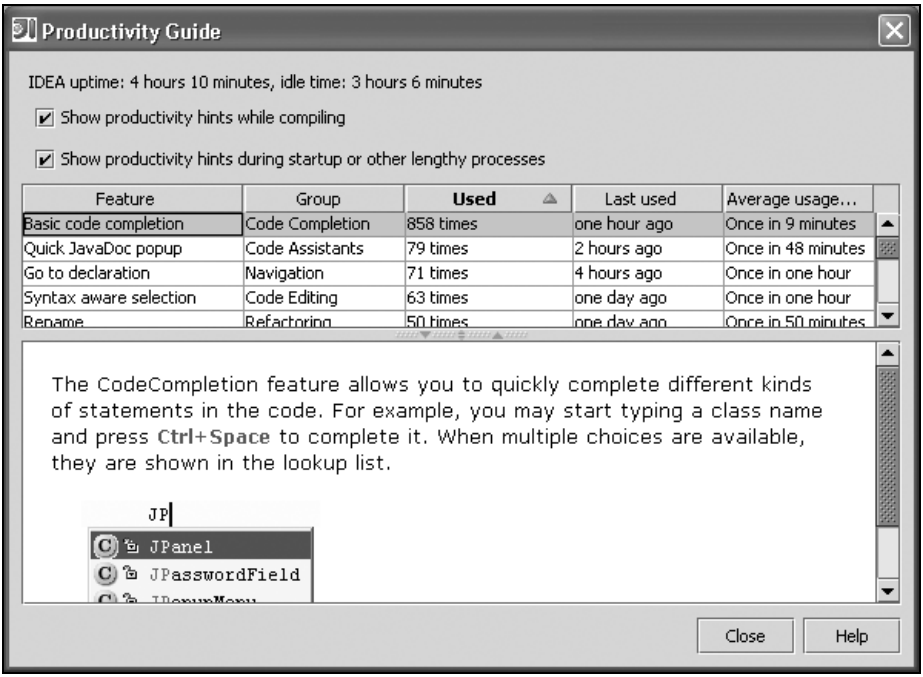


Рис. 6.14. Статистика использования команд

Листинг 6.6

```
import java.util.List;

public class Section {
    private List <Chapter> chapters;

    public Section(List<Chapter> chapters) {
        this.chapters = chapters;
    }

    public List<Chapter> getChapters() {
        return chapters;
    }

    public void setChapters(List<Chapter> chapters) {
        this.chapters = chapters;
```

```
    }
}

public class Chapter {
    private List <Part> parts;
    private String title;

    public Chapter(List<Part> parts, String title) {
        this.parts = parts;
        this.title = title;
    }

    public List<Part> getParts() {
        return parts;
    }

    public void setParts(List<Part> parts) {
        this.parts = parts;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}

public class Part {
    private String title;
    private List<Part> parts;

    public Part(String title) {
        this.title = title;
    }

    public String getTitle() {
```

```
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public List<Part> getParts() {
        return parts;
    }

    public void setParts(List<Part> parts) {
        this.parts = parts;
    }
}
```

Благодаря командам завершения кода, а также встроенным возможностям рефакторинга, можно очень быстро модифицировать код.

Посмотрев на наши классы, мы выделили из них две абстрактные сущности, которые использовали при строительстве остальных классов. В этом нам помог рефакторинг (*см. главу 8*), а также возможности параметризации классов (*см. раздел 11.1*). То, что у нас получилось, представлено в листингах 6.7 и 6.8. Классов стало больше, но сами они стали короче. Из рефакторингов мы использовали **Extract Superclass** (Извлечь базовый класс).

Листинг 6.7

```
public class Titled {
    private String title;

    public Titled(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
```

```
        this.title = title;
    }
}

public class Parted<T> {
    private List<T> parts;

    public Parted(List<T> parts) {
        this.parts = parts;
    }

    public List<T> getParts() {
        return parts;
    }

    public void setParts(List<T> parts) {
        this.parts = parts;
    }
}

public class TitledParted <T> extends Titled {
    private Parted<T> parted;

    public TitledParted(String title, List <T> parts) {
        super(title);
        parted = new Parted<T>(parts);
    }

    public List<T> getParts() {
        return parted.getParts();
    }

    public void setParts(List<T> parts) {
        parted.setParts(parts);
    }
}
```

Листинг 6.8

```
public class Part extends TitledParted <Part> {
    public Part(String title, List<Part> parts) {
        super(title, parts);
    }
}

public class Chapter extends TitledParted<Part> {
    public Chapter(List<Part> parts, String title) {
        super (title, parts);
    }
}

public class Section extends Parted<Chapter> {
    public Section(List<Chapter> chapters) {
        super (chapters);
    }
}

public class Book extends TitledParted<Section>{
    private List<Author> authors;

    public Book(String title, List<Author> authors,
                List<Section> sections) {
        super(title, sections);
        this.authors = authors;
    }

    public List<Author> getAuthors() {
        return authors;
    }

    public void setAuthors(List<Author> authors) {
        this.authors = authors;
    }
}
```

Наплодив такую кучу классов, мы преследовали две цели: сделать код лучше с точки зрения ООП, а также продемонстрировать возможности IntelliJ IDEA при работе с наследованием классов. Давайте возьмем какой-нибудь класс, который мы использовали в наследовании, например `Titled`, и откроем его в редакторе (рис. 6.15).

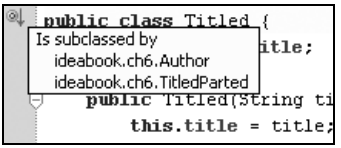


Рис. 6.15. Фрагмент кода класса `Titled` в редакторе

В левом поле редактора рядом с определением класса появился значок с буквой **o** в кружочке со стрелочкой вниз. Этот значок свидетельствует о том, что данный класс имеет производные классы (overridden by). Если подвести к значку курсор мыши, IntelliJ IDEA отобразит всплывающее окошко с информацией о том, какие именно классы являются производными от данного. А если нажать левой кнопкой мыши на значок с буквой **o**, то откроется список всех производных классов с возможностью перехода к ним (рис. 6.16).

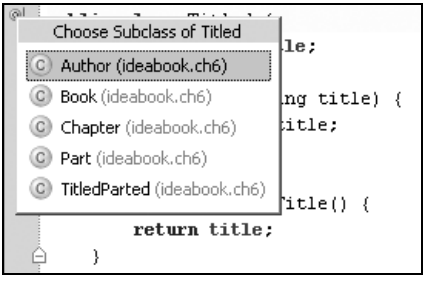


Рис. 6.16. Переход к производному классу

Продолжим демонстрацию возможностей IntelliJ IDEA. Давайте сделаем класс `Parted` перечислимым (т. е. имплементирующим интерфейс `Iterable`). Нам это может потом пригодиться. Допишем к определению класса строку имплементации (рис. 6.17).

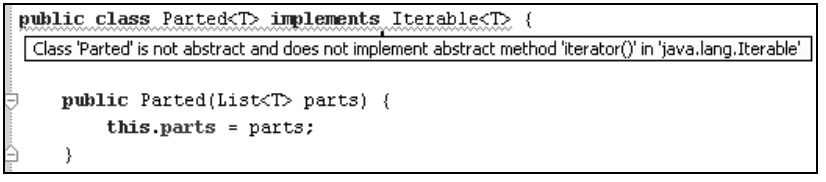


Рис. 6.17. Добавлена строка имплементации

Определение класса стало подчеркнутым красной линией, что свидетельствует об ошибке — в нашем классе пока еще не имплементирован метод интерфейса `Iterable`.

Для его имплементации можно было воспользоваться "подсказкой", которую IntelliJ IDEA покажет, если подвести курсор к ошибке, однако сейчас мы поступим иначе: поместим курсор в тело класса и вызовем команду **Implement Methods** (Имплементировать методы) (<Ctrl>+<I>). Команда отобразит список доступных для имплементации методов (рис. 6.18).

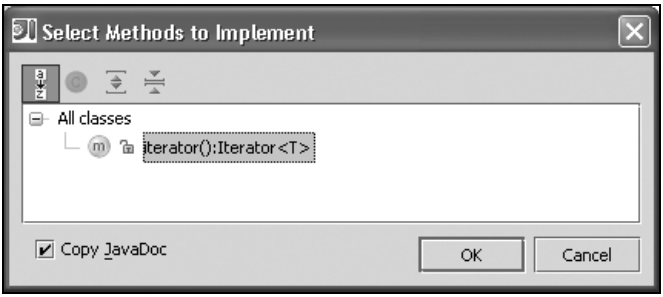
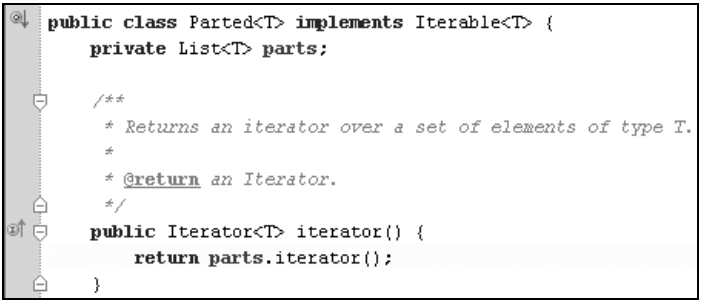


Рис. 6.18. Выбор методов для имплементации

Выбрав методы, мы получим практически готовую имплементацию (рис. 6.19).



```
public class Parted<T> implements Iterable<T> {
    private List<T> parts;

    /**
     * Returns an iterator over a set of elements of type T.
     *
     * @return an Iterator.
     */
    public Iterator<T> iterator() {
        return parts.iterator();
    }
```

Рис. 6.19. Имплементация готова

Обратите внимание на комментарий, IntelliJ IDEA автоматически вставила по нашей просьбе комментарий к созданному методу, скопировав его из метода интерфейса `Iterable`.

Пора подробней остановиться на различных аспектах создания кода, которые мы перечислили в начале этого раздела. Попутно мы также будем дописывать наш код, используя его для примеров к рассматриваемым возможностям.

6.1.1. Завершение кода (completion)

Функция завершения кода позволяет автоматически дописывать различные кодовые конструкции. Всего есть три типа завершения кода:

- простое завершение (basic);
- "умное" завершение типа (smart type);
- завершение имени класса (class name).

Каждое из них следует применять в определенных случаях.

6.1.1.1. Простое (basic) завершение кода

"Горячие" клавиши: <Ctrl>+<Space>.

Данный тип завершения кода используется чаще всего. С его помощью можно завершать ключевые слова, имена переменных, классов, известных в данном контексте, имена методов.

Этого не замечаешь, но когда долго работаешь в IntelliJ IDEA, оказывается, что нажимаешь сочетание клавиш <Ctrl>+<Space> практически постоянно.

Приведем примеры простого завершения кода.

Ключевые слова

Завершению подлежат все ключевые слова, длина которых превышает два символа (см. рис. 6.2). Завершению также подлежат имена простых типов (`int`, `double` и т. п.) (рис. 6.20).

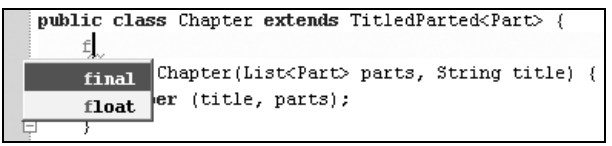


Рис. 6.20. Простое завершение

В зависимости от контекста, в котором была вызвана команда, IntelliJ IDEA может предложить разные варианты завершения.

Имена переменных

Завершение имен переменных работает при определении переменных (когда тип переменной уже записан) (см. рис. 6.13), а также при подстановки имени переменной из текущей области видимости в выражение (рис. 6.21).

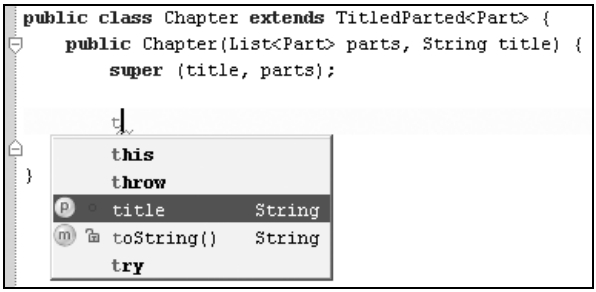


Рис. 6.21. Простое завершение

Имена методов

Завершение имен методов работает аналогично завершению имени переменных. IntelliJ IDEA может предлагать для завершения имена методов из текущей области видимости (см. рис. 6.21).

Имена классов из текущей области видимости

IntelliJ IDEA может выполнять простое завершение имен классов из текущей области видимости. О том, что нужно завершить именно имя класса, IntelliJ IDEA узнает из начальных букв, которые уже набраны в редакторе (рис. 6.22).

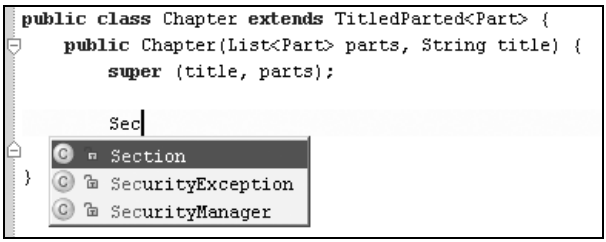


Рис. 6.22. Простое завершение имени класса

В данном случае сыграла роль заглавная буква, с которой написана строка `Sec`. Если бы строка начиналась со строчной буквы '`s`', варианты завершения были бы другими (рис. 6.23).

Рис. 6.23. Простое завершение кода

6.1.1.2. "Умное" завершение типа (smart type)

"Горячие" клавиши: <Ctrl>+<Shift>+<Space>.

Данный тип завершения кода использует для представления вариантов завершения информацию о типе в месте своего применения.

Можно выделить четыре варианта его использования:

- после оператора `return` при возврате результата метода;
- после оператора `new` при создании экземпляра класса;
- после оператора `throw` при выбрасывании исключения;
- при инициализации переменных.

После *return*

Допустим, нам понадобился метод для поиска имени автора книги (класс `Book`) (рис. 6.24).

```
public Author findAuthor (String name) {
    Author foundAuthor = null;

    for (Author author : authors) {
        if (author.getName().equals(name)) {
            foundAuthor = author;
            break;
        }
    }
    return
}
```

Рис. 6.24. Вариант использования "умного" завершения типа

Мы практически написали метод и дошли до возврата значения из него. Если сейчас вызвать простое завершение кода, будет отображен достаточно большой список, среди элементов которого мы, конечно, найдем правильный вариант (переменную `foundAuthor`), однако нам придется потратить на это чуть больше времени, чем если бы мы использовали "умное" завершение типа. Нажав сочетание клавиш <Ctrl>+<Shift>+<Space>, мы сразу получим желаемый результат вместе с точкой с запятой в конце. Метод готов.

После *new*

Нам понадобился еще один метод в классе `Book` для получения итератора по всем составным частям нашей книги до самого нижнего уровня. Мы решили реализовать данный метод следующим образом: собрать в единый список все части, а затем вернуть его итератор. Мы создали метод, а в нем локальную переменную типа `List<Part>`, которую собрались проинициализировать списком `ArrayList<Part>` (рис. 6.25).

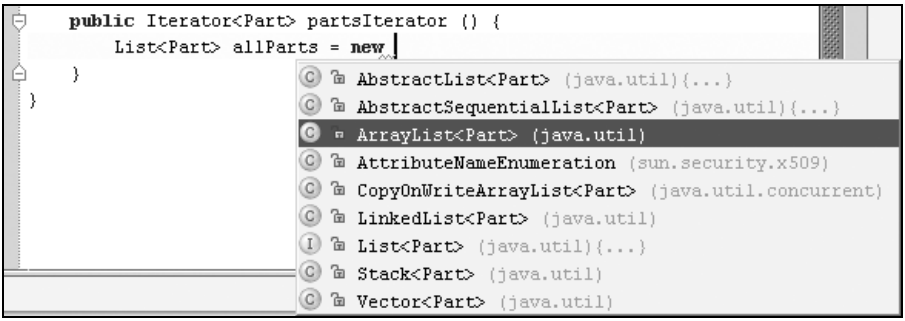


Рис. 6.25. Вызов "умного" завершения типа после оператора `new`

Мы не стали вручную писать класс инициализации, а вызвали команду "умного" завершения типа. Данная команда на основе имеющейся информации о типе инициализируемого нами параметра вывела список доступных типов для инициализации, из которых мы и выбрали требуемый.

После *throw*

Мы решили запретить возможность клонирования объекта типа `Book`, для чего мы переопределим его метод `clone()`, единственным действием которого будет выбрасывание исключения `CloneNotSupportedException` (рис. 6.26).

Рис. 6.26. "Умное" завершение типа исключения

После операторов `throw` и `new` мы вызвали команду завершения кода <Ctrl>+<Shift>+<Space>, и IntelliJ IDEA представила нам на выбор два варианта, один из которых мы и выбрали.

При инициализации

Данный вариант "умного" завершения типа работает, если в классе инициализируемого объекта имеются его константные значения либо методы фабрики.

Мы решили добавить в класс `Book` поле для указания издательства, в котором книга была издана. Для описания издательства, мы использовали конструкцию `enum` — листинг 6.9.

Листинг 6.9

```
public enum PublishingHouse {
    BHV("BHV"), PITER("Piter"), NEW_READERS("New Readers");
```

```
    private final String name;

    PublishingHouse(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Для тестирования данного класса мы решили создать в нем метод `main`, в котором выполним простейшую проверку (рис. 6.27).

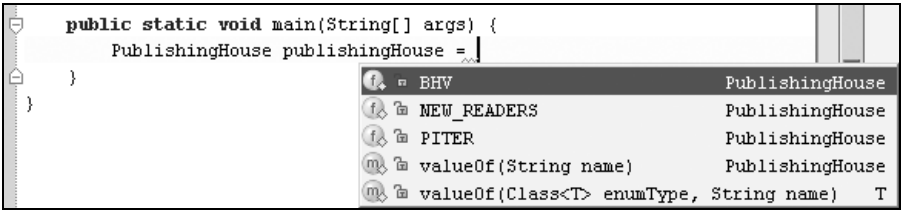


Рис. 6.27. "Умное" завершение типа при инициализации

После знака = мы вызвали команду "умного" завершения типа, которая выдала нам список возможных вариантов инициализации.

6.1.1.3. Имя класса (class name)

Горячие клавиши: <Ctrl>+<Alt>+<Space>.

В отличие от предыдущих двух типов завершения кода, у этого есть только один вариант применения. Завершение имени класса выдает список возможного имени класса, учитывая не только классы из области видимости (как это делает простое завершение кода), но также и все подходящие имена классов из всех доступных библиотек, т. е. даже если имени класса нет в списке импортов (рис. 6.28).

Мы решили добавить возможность сериализации класса `PublishingHouse`, для чего включили в список интерфейсов, которые он имплементирует, интерфейс `Serializable`. Написав первые три буквы названия интерфейса, мы вызвали команду завершения имени типа.

После того как требуемый интерфейс был выбран из списка, IntelliJ IDEA автоматически добавила `java.io.Serializable` в строку импорта.

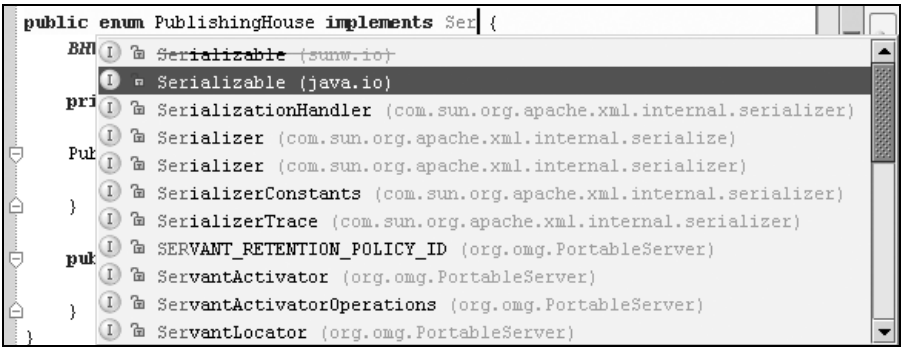


Рис. 6.28. Завершение имени класса

6.1.2. Подсветка

Мы уже успели немного рассказать о подсветке в редакторе IntelliJ IDEA. Можно выделить три ключевых типа подсветки:

- подсветка кода, к которой можно отнести выделение ключевых слов, статических методов, строк и т. п. Полный перечень пунктов подсветки кода можно найти в *разделе 4.1.6.2*;
- подсветка скобок: фигурных, круглых и квадратных;
- подсветка ошибок.

6.1.2.1. Код

Примеров подсветки кода можно привести очень много, достаточно посмотреть на иллюстрации с фрагментами кода в редакторе IntelliJ IDEA, например, рис. 6.2, 6.9, 6.24.

Функция подсветки кода заключается в специальном цветовом или шрифтовом выделении отдельных элементов кода. Так, например, ключевые слова (по умолчанию) выделяются жирным шрифтом и пишутся буквами темно-синего цвета. Статические поля и методы записываются наклонным шрифтом. Поля класса — жирным шрифтом. Строки — жирным шрифтом, буквами зеленого цвета и т. п.

Подсветка кода очень помогает ориентироваться в коде и экономит время. Зачастую даже без углубленного изучения контекста можно сказать, что вот это локальная переменная, а это экземпляр какого-то класса.

6.1.2.2. Скобки

Подсветка скобок также весьма удобна. В Java используются скобки четырех типов (если считать параметризованные классы), IntelliJ IDEA умеет подсвечивать парные фигурные `{}`, круглые `()` и квадратные `[]` скобки.

Для скобок происходит еще и дополнительная маркировка в левом поле (рис. 6.29—6.31).

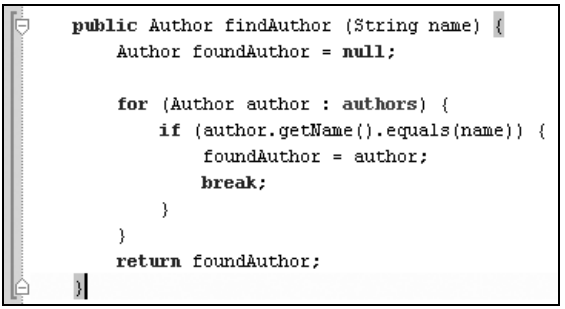


Рис. 6.29. Дополнительная маркировка при подсветке фигурных скобок

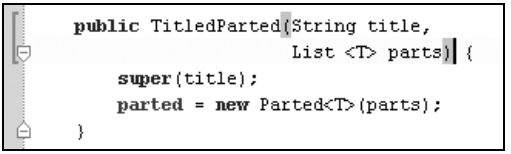


Рис. 6.30. Дополнительная маркировка при подсветке круглых скобок

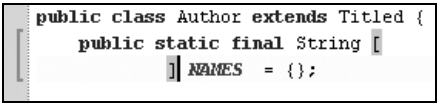


Рис. 6.31. Дополнительная маркировка при подсветке квадратных скобок

6.1.2.3. Ошибки

IntelliJ IDEA умеет находить и подсвечивать два типа ошибок:

- синтаксические ошибки — те, которые препятствуют компиляции Java-кода — просто ошибки;
- логические ошибки — конструкции, правильные с точки зрения синтаксиса языка, но которые, тем не менее, либо не имеют смысла, либо могут привести к возможным ошибкам — замечания.

Ошибки выделяются в редакторе красным шрифтом и/или красным волнистым подчеркиванием (рис. 6.32).

Если подвести к подсветке ошибки либо к маркеру в правом поле редактора курсор мыши, IntelliJ IDEA отобразит сообщение о причине ошибки.

Логические ошибки обнаруживаются специальными функциями проверки кода — инспекциями. Подробнее об инспекциях можно прочитать в *разде-*

ле 6.4.1. Логические ошибки выделяются желтыми маркерами в правом поле редактора. Код, который содержит логическую ошибку, часто имеет серый цвет символов (см. рис. 6.4).

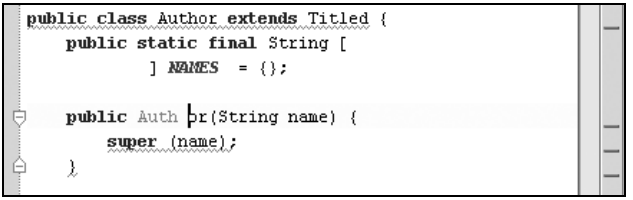


Рис. 6.32. Подсветка ошибок

6.1.3. "Подсказки" (intention actions)

"Подсказки" — это визуальный механизм оповещения программиста о том, что с кодом, который он пишет, можно выполнить некоторые действия. Вариантов таких действий довольно много. Так, в самом начале *раздела 6.1* мы использовали "подсказку" для создания методов `getTitle()` и `setTitle()` — для полей класса (см. рис. 6.4).

6.1.3.1. Настройка "подсказок"

Настроить отображение тех или иных "подсказок" можно следующими способами:

- для "подсказок", поставляемых в комплекте с плагином IntentionPowerPack (*см. раздел 4.1.21*), имеется специальный диалог в настройках IntelliJ IDEA, в котором можно включить те или иные подсказки;
- "подсказки", генерируемые инспекциями (*см. разделы 4.1.5, 6.4.1*), можно настроить с помощью диалога **Errors** (Ошибки) в настройках IntelliJ IDEA;

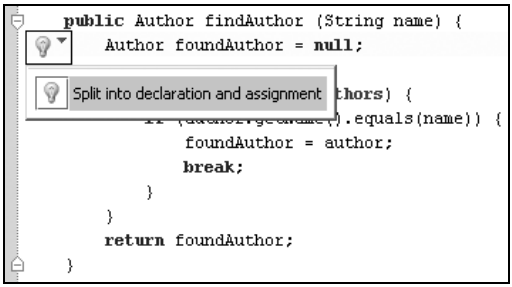


Рис. 6.33. Выключатель подсказки

- встроенные "подсказки" IntelliJ IDEA приходится настраивать поштучно. При раскрытии перечня действий "подсказок" значки с лампочкой служат своеобразным выключателем для "подсказки" (рис. 6.33). Если на

значок с лампочкой нажать — лампочка погаснет, и соответствующая "подсказка" не будет автоматически появляться. Чтобы включить ее обратно, или чтобы посмотреть возможные "подсказки" по данному коду, можно использовать сочетание клавиш <Alt>+<Enter>. Таким же образом можно настроить и все остальные типы "подсказок".

Примечание

В следующей версии IntelliJ IDEA появится возможность расширенного управления всеми "подсказками" в системе.

В IntelliJ IDEA есть два типа "подсказок":

- "подсказки" изменения кода;
- "подсказки" исправления кода.

Первый тип "подсказок" активизируется, когда с кодом все в порядке, по крайней мере, с точки зрения языка (значок — желтая лампочка). Второй тип — эти "подсказки" еще называют "корректировкой" или "Quick Fix" — появляются, когда в коде есть синтаксические ошибки (значок — лампочка с восклицательным знаком).

На "подсказки" первого типа вполне можно не обращать внимания, если вы точно знаете, что код должен быть именно таким, какой он есть. "Подсказки-корректировки" лучше изучать более внимательно.

Далее мы приведем полный перечень всех "подсказок" IntelliJ IDEA. "Подсказки", поставляемые вместе с плагином IntentionPowerPack, не включены в данный список, поскольку они подробно описаны в *разделе 4.1.21*.

В описании "подсказок" мы будем приводить их английское название — то, с которым они встречаются в интерфейсе IntelliJ IDEA, а также краткое описание действия подсказки. Если у "подсказки" нет фиксированного названия, мы будем просто приводить ее описание.

6.1.3.2. "Подсказки" для Java-кода

В IntelliJ IDEA имеются следующие подсказки об изменении кода:

- **Add Method Body** (Добавить тело метода) — добавить тело по умолчанию неабстрактному методу, у которого нет тела;
- **Add static imports** (Добавить статический импорт) — добавить статический импорт для статических классов и членов класса (для Java 5);
- **Add @Override annotation** (Добавить аннотацию `@Override`) — добавить аннотацию `@Override` методу, который переопределяет метод из базового класса (для Java 5);
- **Add return statement** (Добавить оператор `return`) — добавить методу оператор `return` со значением по умолчанию, если метод возвращает значение, а оператор `return` пропущен;

- **Add type cast** (Добавить приведение типа) — добавить необходимое приведение типа в операторе присваивания, операторе `return`, инициализации переменной, операторе `switch`, инициализации массива;
- **Adjust package name** (Скорректировать имя пакета) — привести имя пакета класса в соответствие с физическим путем на диске;
- **Delete catch** (Удалить catch-блок) — удалить неиспользуемый или излишний catch-блок в операторе `try-catch`;
- **Implement methods** (Имплементировать методы) — имплементировать методы класса;
- **Fix Method return** (Исправить оператор `return` метода) — исправить тип возвращаемого значения метода в соответствии с типом выражения оператора `return`;
- **Fix method throws** (Исправить список выбрасываемых исключений метода) — скорректировать список выбрасываемых исключений метода в соответствии со списком метода из базового класса, который он переопределяет;
- **Move catch up** (Переместить catch-блок выше) — переместить catch-блок исключения выше по коду, чтобы избежать проблемы перекрытия данного исключения другим, например, переместить catch-блок исключения `IOException` перед catch-блоком исключения `Exception`;
- **Move class to separate file** (Переместить класс в отдельный файл) — переместить класс верхнего уровня с модификатором `public` в файл с именем, соответствующим имени класса (ClassName.java);
- **Move to package** (Переместить в пакет) — переместить класс, имеющий описание пакета, несоответствующее физическому пути, в пакет, соответствующий описанию;
- **Must be final fix** (Должно иметь модификатор `final`) — исправляет ситуацию, когда к переменной, не имеющей модификатора `final`, происходит обращение из внутреннего класса. Исправление происходит одним из трех перечисленных образов:
  - переменная объявляется как `final`;
  - создается новая переменная с модификатором `final`, которой присваивается значение используемой переменной;
  - создается массив с модификатором `final` из одного элемента — значения используемой переменной, и этот элемент массива будет применен для доступа к переменной;
- **Optimize imports** (Оптимизировать импорты) — выполнить команду **Optimize Imports** (Оптимизировать импорты) при обнаружении неиспользуемой строки импорта (*см. раздел 3.2.11.2*);

- **Remove redundant cast** (Убрать излишнее приведение типа) — убрать излишнее приведение типа;
- **Super method return fix** (Скорректировать оператор `return` метода из базового класса) — заменить тип возвращаемого значения и оператор `return` из базового класса в соответствии с типом выражения, возвращаемого методом в производном классе;
- **Surround with try catch** ("Обернуть" блоком try-catch) — "обернуть" фрагмент кода, выбрасывающий исключения, блоком try-catch;
- **Variable type fix** (Исправить тип переменной) — исправить тип переменной в соответствии с типом выражения, которое ее инициализирует;
- **Add exception to throws** (Добавить исключение в список `throws`) — добавить необрабатываемое исключение в список выбрасываемых исключений метода;
- **Create field from parameter** (Создать поле класса из параметра) — создать поле класса из неиспользуемого параметра вызова конструктора;
- **Create class from new** (Создать класс из оператора `new`) — создать класс на основе информации использования оператора `new`. Например, если имеется следующий код `X x = new X (10.5, "abc");`. При этом класс `X` не определен, IntelliJ IDEA создаст класс `X` с конструктором, который был использован в первоначальном выражении;
- **Create class from usage** (Создать класс из обращения к нему) — создать класс на основе информации об обращении к нему;
- **Create constructor from new** (Создать конструктор из оператора `new`) — создать конструктор класса на основе информации из вызова оператора `new` для несуществующего конструктора;
- **Create field from usage** (Создать поле из обращения к нему) — создать поле, основываясь на информации об обращении к нему;
- **Create local from usage** (Создать локальную переменную из обращения к ней) — создать локальную переменную, основываясь на информации об обращении к ней;
- **Create parameter from usage** (Создать параметр из обращения к нему) — создать параметр вызова метода, основываясь на информации об обращении к нему;
- **Create method from usage** (Создать метод из обращения к нему) — создать метод, основываясь на информации об обращении к нему;
- **Create getter/setter from usage** (Создать методы `get`/`set` из обращения к ним) — создать методы `get()` и `set()`, основываясь на информации об обращении к ним;

- **Implement abstract method** (Имплементировать абстрактный метод) — имплементировать абстрактный метод в одном из соответствующих классов;
- **Rename Reference** (Переименовать ссылку) — переименовать все ссылки неизвестной переменной или класса, именем, доступным в пределах данного контекста;
- **Invert if condition** (Перевернуть условный оператор `if`) — поменять местами ветки условного оператора вместе с изменением условия;
- **Split declaration** (Разделить определение) — разделить определение переменной с инициализацией на отдельное определение и отдельную инициализацию. Также разделить определение нескольких переменных одного типа в одной строке;
- **Split if action** (Разделить оператор `if`) — разделить оператор `if`, в условии которого используется операция логического 'и' (`&&`) или логического 'или' (`||`), на несколько операторов `if`;
- **Add exception to catch** (Добавить обработку исключения) — добавить catch-блок для необрабатываемого исключения;
- **Implement interface/abstract class** (Имплементировать интерфейс/абстрактный класс) — создать имплементацию указанного интерфейса или абстрактного класса;
- **Make [...] not abstract** (Сделать класс не абстрактным) — при попытке создания экземпляра абстрактного класса, у которого нет абстрактных методов;
- **Make [...] abstract** (Сделать абстрактным) — сделать класс, у которого есть абстрактные методы, абстрактным;
- **Make [...] not final** (Убрать модификатор `final`) — убрать модификатор `final` у класса, при попытке наследования от него;
- **Make [...] not public** (Убрать модификатор `public`) — убрать модификатор доступа `public` у класса верхнего уровня, если его имя не соответствует имени файла, в котором он находится;
- **Make [...] not static** (Убрать модификатор `static`) — убрать модификатор `static` у статического поля/метода/инициализации класса, объявленной во внутреннем не статическом классе;
- **Make [...] static** (Сделать внутренний класс статическим) — сделать внутренний класс со статическим полем/методом/инициализатором класса, либо без явного доступа к не статическим методам/полям класса верхнего уровня, статическим;
- **Make [...] not final** (Убрать модификатор `final`) — убрать модификатор `final` у метода из базового класса при попытке переопределить этот метод;

- **Make [...] abstract** (Сделать абстрактным) — сделать метод абстрактным, когда его тело отсутствует;
- **Make [...] not static/ Make [...] static** (Убрать модификатор `static` у метода/Сделать метод статическим) — сделать переопределенный метод базового класса (не)статическим, если переопределяющий метод в производном класса (не)статический;
- **Make [...] not static/ Make [...] static** (Убрать модификатор `static` у метода/Сделать метод статическим) — сделать переопределяющий метод в производном классе (не)статическим, если переопределенный метод базового класса (не)статический;
- **Make [...] public/Make [...] package local** (Установить модификатор доступа) — установить для поля, метода, класса соответствующий модификатор доступа, при попытке получить доступ из другого контекста, до этого закрытого;
- Удалить конфликтующие модификаторы (например, `abstract final`);
- **Make [...] not public/protected/private/volatile/transient** (Убрать модификатор доступа) — удалить недопустимый модификатор (например, `private`-модификатор в локальном классе);
- **Make [...] static** (Сделать статическим) — сделать поле/метод/класс статическим при попытке доступа к нему из статического контекста.

6.1.3.2. "Подсказки" для XML

В IntelliJ IDEA имеются следующие подсказки для XML:

- **Fetch external resource** (Скачать внешний ресурс) — скачать внешний ресурс, ссылка на который имеется в XML-файле (*см. раздел 4.1.10*);
- **Ignore external resource** (Игнорировать внешний ресурс) — пометить ресурс, ссылка на который имеется в XML-файле, как игнорируемый (*см. раздел 4.1.10*).

Подробнее про поддержку XML в IntelliJ IDEA можно прочитать в *разделе 10.1.*

6.1.3.3. Подсказки для Ant

В IntelliJ IDEA имеются следующие подсказки для Ant:

- создать цель, если используется цель с несуществующим именем;
- создать свойство, если используется свойство с несуществующим именем.

6.1.4. Форматирование кода

Когда мы учились программированию, было популярным обсуждение темы "правильного" оформления кода — с определенными выравниваниями, рас-

становкой скобок, пробелов и т. п. Помним, как ругались бывалые программеры на "некрасиво" оформленный код.

Когда для создания кода используется IntelliJ IDEA, проблемы с форматированием исчезают. Дело в том, что IntelliJ IDEA обладает встроенными возможностями форматирования кода в соответствии со стилевыми настройками, которые можно задать глобально (*см. раздел 4.1.7*), а также индивидуально для каждого проекта (*см. раздел 4.2.4*). Даже если вам приходится редактировать код, форматирование которого вызывает стойкое чувство отвращения, буквально одним нажатием клавиш можно переформатировать его в соответствии с вашими предпочтениями.

В IntelliJ IDEA есть две функции форматирования кода:

- **Reformat Code** (Переформатировать код) (<Ctrl>+<Alt>+<L>) — переформатировать код в соответствии с пользовательскими настройками (*см. раздел 3.2.11.1*);
- **Auto Indent Lines** (Автоматически выровнять строки) (<Ctrl>+<Alt>+<I>) — установить отступ для строк кода.

Переформатирование кода может быть выполнено для отдельного файла или всего проекта, а выравнивание строк применяется только к текущему файлу.

6.1.5. Информация о коде

При написании кода IntelliJ IDEA может отображать для вас дополнительную информацию о коде, который вы пишите. К такой информации можно отнести:

- комментарии JavaDoc для методов и классов, которыми вы оперируете. Для отображения комментариев можно использовать команду **Quick JavaDoc** (Быстрый JavaDoc) (<Ctrl>+<Q>). Данная команда работает не только в самом редакторе, но также и в любом другом элементе интерфейса IntelliJ IDEA, где могут встретиться имя класса или метода, например, при переходе к классу (<Ctrl>+<N>). Также *см. раздел 3.2.4.7*;
- информация о параметрах вызова метода. При написании вызова метода IntelliJ IDEA может отображать информацию о его параметрах. Для отображения этой информации можно использовать команду **Parameter Info** (Информация о параметрах) (<Ctrl>+<P>). Также *см. раздел 3.2.4.8*;
- информация о контексте. Часто при редактировании объемных исходных файлов можно забыть, в каком контексте (в каком классе, методе и т. п.) вы находитесь в данный момент. Чтобы не тратить время на прокрутку окна и просмотр кода, в IntelliJ IDEA предусмотрена специальная команда для просмотра контекста **Context Info** (Информация о контексте) (<Alt>+<Q>). Также *см. раздел 3.2.4.9*.

6.1.6. Буфер обмена

В IntelliJ IDEA по сравнению с обычным текстовым редактором имеются некоторые особенности при работе с буфером обмена. Кроме стандартных операций копирования (*см. раздел 3.2.2.4*), удаления в буфер обмена (*см. раздел 3.2.2.3*) и вставки (*см. раздел 3.2.2.6*) в IntelliJ IDEA умеет запоминать несколько объектов, помещенных в буфер обмена, и при вставке позволять выбирать необходимый объект (*см. раздел 3.2.2.7*).

При копировании кода IntelliJ IDEA может автоматически его выравнивать в зависимости от контекста копирования.

Если копируются строки, IntelliJ IDEA может автоматически заменять служебные символы вроде перевода каретки на соответствующие последовательности. Допустим, мы скопировали фрагмент кода, представленный на рис. 6.34.

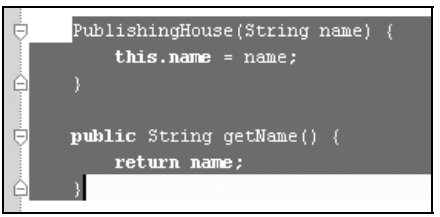


Рис. 6.34. Скопирован фрагмент кода

Если теперь мы скопируем этот код внутрь строки, IntelliJ IDEA автоматически сделает так, чтобы эта строка была корректной с точки зрения синтаксиса Java (рис. 6.35).

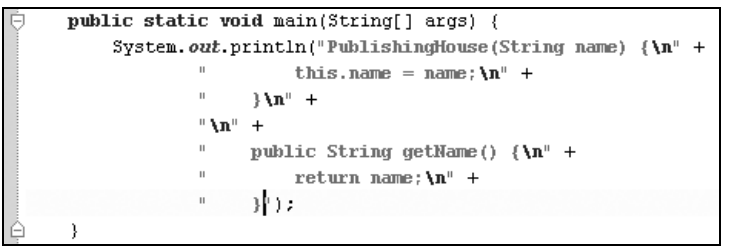


Рис. 6.35. Фрагмент скопирован внутрь строки

Кроме обычного текста, команды операций с буфером обмена работают также и с файлами.

6.1.7. Сокращенное представление кода

Сокращенное представление кода или "Folding" (в терминах IntelliJ IDEA) — это возможность работать с кодом в редакторе IntelliJ IDEA по аналогии с иерархическим списком или деревом.

Структурные элементы или блоки кода могут быть свернуты и развернуты в соответствии с пожеланием программиста.

Сокращенное отображение кода очень удобно при просмотре или анализе кода, когда вам не требуется видеть код каких-то методов, и вы просто их сворачиваете (рис. 6.36).

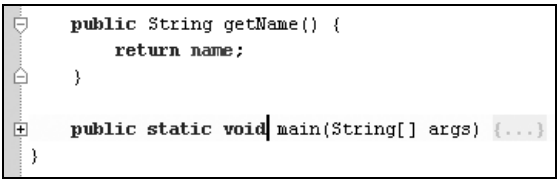


Рис. 6.36. Свернутое представление кода

Свернутый код отображается в виде символа многоточия с серым фоном. Также на полях редактора имеются специальные значки управления видимостью отдельных структурных участков кода (*см. раздел 3.2.4.19*).

6.2. Навигация по коду

IntelliJ IDEA предоставляет множество различных способов навигации по коду — между файлами, классами, методами и т. п.

Для каждой сущности, которой умеет оперировать IntelliJ IDEA, существует как минимум одна, а чаще несколько команд, для перехода.

Есть универсальные команды, которые удобно использовать в любом контексте. Такие команды нужно просто помнить наизусть. Также многие команды могут быть продублированы с помощью некоторых действий с текущим контекстом, причем сделано это может быть намного проще и быстрее, чем с использованием универсальных команд.

В этом разделе мы расскажем об универсальных командах навигации.

6.2.1. Класс

Переход к конкретному классу — команда, выполняемая относительно часто. Есть множество способов перейти к конкретному классу, самым удобным и простым из которых является команда **Go To | Class** (Перейти | К классу) (<Ctrl>+<N>). С помощью этой команды можно перейти к произвольному классу как внутри проекта, так и к классу из библиотеки вне зависимости от контекста вызова данной команды. Также *см. раздел 3.2.5.1.*

6.2.2. Файл

Команда перехода к файлу позволяет открыть в редакторе любой текстовый файл известного типа (*см. раздел 4.1.11*). Открытие файла происходит ана-

логично открытию класса — с помощью специальной команды **Go To | File** (Перейти | К файлу) (<Ctrl>+<Shift>+<N>). Также *см. раздел 3.2.5.2.*

6.2.3. Метод или поле

Переход к полю или методу класса можно осуществить с помощь команды, аналогичной командам перехода к классу или файлу — **Go To | Symbol** (Переход | К символу) (<Ctrl>+<Shift>+<N>) (*см. раздел 3.2.5.3*).

Кстати, между командами перехода к классу, файлу и символу можно легко переключаться, при этом фрагмент искомого имени не будет потерян.

6.2.4. Переход к определению, имплементации, определению типа или базовому методу

Переход к имплементации, определению типа или базовому методу осуществляется с помощью специальных пиктограмм, расположенных в левом поле окна редактора (рис. 6.37).

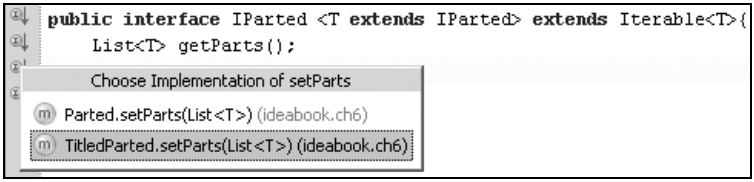


Рис. 6.37. Переход к имплементации

Если класс имеет производные классы, базовый класс (отличный от `java.lang.Object`) или имплементирует интерфейсы, рядом с именем класса или напротив соответствующих методов в левом поле редактора появится соответствующая пиктограмма. Если на эту пиктограмму нажать левой кнопкой мыши, можно либо сразу перейти к указанному классу или методу, либо выбрать этот класс или метод из списка.

6.2.5. Перемещение по коду "вперед-назад"

В IntelliJ IDEA имеется очень удобная возможность перемещаться в редакторе между теми файлами и позициями в этих файлах, которые вы посещали до этого — своеобразный аналог кнопок **Back/Next** (Назад/Вперед) в браузере.

Такое перемещение можно осуществлять с помощью команд **Go To | Back/Next** (Перейти | Назад/Вперед) (<Ctrl>+<Alt>+<Left>/<Right>).

6.2.6. Список последних файлов

Очень часто бывает, что при редактировании кода закроешь какой-нибудь файл, а потом снова в него нужно заглянуть. IntelliJ IDEA запоминает последние открытые файлы и позволяет быстро открыть один из таких файлов с помощью команды **Recent Files** (Последние файлы) (<Ctrl>+<E>) (*см. раздел 3.2.4.14*).

6.2.7. Закладки

Если вам приходится возвращаться к какому-либо месту в файле достаточно часто, на него можно установить закладку. В IntelliJ IDEA есть два типа закладок — обычные и нумерованные. Удобнее всего пользоваться нумерованными закладками, потому что для перехода к ним достаточно одного нажатия клавиш (*разделы 3.2.2.12—3.2.2.13*).

6.2.8. Заметки (TODO)

IntelliJ IDEA умеет выводить список комментариев в коде, оформленных некоторым специальным образом (*см. раздел 4.1.18*).

Такие комментарии отображаются в едином списке, с помощью которого можно легко переходить между файлами, содержащими эти комментарии.

6.2.9. Иерархия

Часто бывает нужно видеть сразу всю иерархию наследования текущего класса или цепочку вызова методов. IntelliJ IDEA имеет несколько команд, позволяющих легко (легко для программиста, но не всегда легко для процессора — операции отображения иерархии часто занимают длительное время) отображать три типа иерархии:

- иерархию классов (или интерфейсов) — команда **View | Type Hierarchy** (Просмотр | Иерархия типа) (<Ctrl>+<H>) (*см. раздел 3.2.4.16*);
- иерархию методов — команда **View | Method Hierarchy** (Просмотр | Иерархия метода) (<Ctrl>+<Shift>+<H>) (*см. раздел 3.2.4.17*);
- иерархию вызова методов — команда **View | Call Hierarchy** (Просмотр | Иерархия вызова) (<Ctrl>+<Alt>+<H>).

Все эти команды отображают соответствующую иерархию в инструментальном окне **Hierarchy** (Иерархия) (*см. раздел 3.4.8.9*).

6.3. Генерация кода

IntelliJ IDEA обладает богатыми возможностями автоматического создания однотипного кода. К числу таких типов кода можно отнести следующие:

- классы, интерфейсы, перечисления;
- блоки кода для Java-операторов;
- файлы специального типа;
- методы класса;
- комментарии;
- строки импорта.

Благодаря возможностям генерации, создавать код в IntelliJ IDEA можно очень быстро и очень качественно.

6.3.1. Шаблоны файлов

Для создания однотипных файлов, например классов, интерфейсов, перечислений (`enum`), специальных классов вроде EJB и прочих, используются так называемые шаблоны файлов (*см. раздел 4.1.14*).

С помощью этих шаблонов можно настроить, каким будет "скелет" создаваемых сущностей, какие комментарии будут в нем, какие обязательные элементы он будет содержать и т. п.

6.3.2. "Живые шаблоны"

"Живые шаблоны" позволяют превращать короткую символьную последовательность — аббревиатуру, во фрагмент Java-кода. Есть очень простые шаблоны, например, для генерации простого кода, вроде перечня модификаторов метода или поля класса. Есть более сложные, которые позволяют создавать код на основе имеющего контекста.

Подробнее о "живых шаблонах" рассказано в *разделе 4.1.13*.

6.3.3. Создание однотипных классов, интерфейсов и перечислений

IntelliJ IDEA умеет создавать классы, интерфейсы и перечисления либо по специальной команде **Edit | New** (Редактирование | Создать) (*см. раздел 3.2.2.10*), либо при выполнении действий по различным подсказкам (*см. раздел 6.1.3*).

Класс создается на основе соответствующего шаблона файла (*см. раздел 4.1.14.1*).

6.3.4. Создание однотипных файлов

Если вам требуется часто создавать однотипные файлы, например XML-файлы определенной структуры, отчеты, другие типы файлов, можно создать собственные шаблоны файлов (*см. раздел 4.1.14*) для создания соответствующих файлов.

6.3.5. Создание методов *get* и *set*

Если у вас есть класс, для полей которого нужно создать методы `get` и `set`, это вполне можно поручить сделать IntelliJ IDEA. IntelliJ IDEA сама проанализирует структуру класса и предложит вам на выбор поля, для которых можно сгенерировать соответствующие методы (*см. раздел 3.2.6.4*).

6.3.6. Автоматическое создание конструкторов

Аналогично созданию методов `get` и `set` (*см. раздел 6.3.6*), IntelliJ IDEA автоматически создает конструкторы для класса, учитывая все необходимые поля и конструкторы базовых классов (если таковые имеются) (*см. раздел 3.2.6.4*).

6.3.7. Создание методов

IntelliJ IDEA умеет генерировать следующие типы методов:

- методы, имплементирующие (*см. раздел 3.2.6.2*) или переопределяющие (*см. раздел 3.2.6.1*) методы базового класса;
- имплементацию методов `equals()` и `hashCode()` для заданного класса (*см. раздел 3.2.6.4*);
- прочие методы, создаваемые при работе "подсказок".

6.3.8. Автоматическое создание блоков кода

Часто при создании кода требуется создавать стандартные конструкции вроде условных операторов, циклов, итераций по коллекциям, операторов `try-catch-finally` и т. п.

Вместо того, чтобы описывать такие конструкции вручную, можно воспользоваться возможностями IntelliJ IDEA для их автоматического создания.

Если у вас есть некоторый блок кода (возможно пустой), который требуется поместить внутрь некоторого оператора, можно использовать команду IntelliJ IDEA **Code | Surround With** (Код | Обернуть в) (<Ctrl>+<Alt>+<T>) (*см. раздел 3.2.6.3*).

С помощью этой команды можно завернуть код в:

- условный оператор `if` или `if-else`;
- циклы `while`, `do-while`, `for`;
- блоки обработки исключений `try`, `try-catch`, `try-finally`, `try-catch-finally`;
- блок синхронизации `synchronized`.

Более "продвинутые" методы генерации блоков кода предоставляют "живые шаблоны" (*см. раздел 4.1.13*).

6.3.9. Строки импорта

В IntelliJ IDEA имеется замечательная функция автоматического добавления строк импорта. С тех пор, как мы перешли на IntelliJ IDEA, мы ни разу не записали оператор импорта вручную.

IntelliJ IDEA может вставлять операторы импорта при возникновении использования какого-либо класса, которого нет в текущей области видимости, при копировании фрагментов кода, который использует классы, "неизвестные" в текущем контексте.

Также имеется возможность автоматически оптимизировать строки импорта (*см. раздел 3.2.11.2*) в соответствии с пользовательскими настройками (*см. раздел 4.1.7*).

6.3.10. Комментарии

Если в теле класса создать комментарий в стиле JavaDoc (рис. 6.38), IntelliJ IDEA автоматически создаст максимум JavaDoc-тегов для описания указанной сущности (рис. 6.39).

```
/**
public TitledParted(String title, List <T> parts) {
    super(title);
    parted = new Parted<T>(parts);
}
```

Рис. 6.38. Создание комментария JavaDoc

```
/**
 *
 * @param title
 * @param parts
 */
public TitledParted(String title, List <T> parts) {
    super(title);
    parted = new Parted<T>(parts);
}
```

Рис. 6.39. Результат создания комментария JavaDoc

То, что вы можете увидеть на рис. 6.39, IntelliJ IDEA сделала сама.

Кроме параметров, IntelliJ IDEA также умеет генерировать JavaDoc-теги для возвращаемого значения и выбрасываемых исключений.

6.4. Анализ кода

IntelliJ IDEA обладает колоссальными возможностями анализа кода. В *разделе 6.1.3* мы уже рассказывали об одной из сторон анализа кода — "подсказках", помогающих быстро видоизменять и корректировать код.

Кроме них в IntelliJ IDEA имеются следующие возможности анализа кода:

- инспекции — специальные функции проверки кода, которые могут быть выполнены автоматически или по запросу пользователя;
- анализ зависимостей в коде;
- анализ дубликатов кода.

Далее мы рассмотрим указанные типы анализа кода.

6.4.1. Проверка кода (inspections)

IntelliJ IDEA обладает мощными встроенными возможностями по распознаванию ошибок в коде программы еще до того, как она будет скомпилирована и запущена. Кроме синтаксических ошибок — тех, что может найти и компилятор, IntelliJ IDEA позволяет находить различные недочеты, которые хоть и являются верными с точки зрения синтаксиса языка, однако могут приводить к разным неприятным последствиям, либо просто являются лишними.

Найденные ошибки и недочеты в коде IntelliJ IDEA выделяет цветом и отмечает на полях редактора горизонтальными черточками (рис. 6.40).

Рис. 6.40. Подсветка ошибок в редакторе

Так как в IntelliJ IDEA имеется более 300 инспекций, для удобства обращения с ними, инспекции объединены в группы. Всего существует 26 групп инспекций, которые мы подробно рассмотрим в *разделах 6.4.4.2—6.4.4.26*.

6.4.4.1. Запуск и настройка

Есть три варианта запуска инспекций.

- Автоматический запуск. Если в настройках проверки ошибок (*см. раздел 4.1.5*) настроить для соответствующих пунктов в списке ошибок уровень **Warning** (Предупреждение) или **Error** (Ошибка), такие ошибки будут автоматически помечаться в коде. Ошибки, в данном случае, соответствуют той или иной инспекции, которая умеет их находить.
- Запуск вручную. В IntelliJ IDEA имеется специальная команда **Analyze | Inspect Code** (Анализ | Инспекция кода) (*см. раздел 3.2.7.1*), с помощью которой можно запустить указанные типы инспекций на весь проект или конкретный класс.
- В больших проектах инспекции могут работать достаточно долго, поэтому в IntelliJ IDEA имеется возможность инспектировать код отдельно от его написания с помощью специальной утилиты inspect, входящей в поставку IntelliJ IDEA (*см. раздел 2.2*). Результаты работы этой утилиты заносятся в специальный файл, который затем можно открыть в IntelliJ IDEA и просматривать результаты инспекций так, как если бы они были выполнены внутри самой IntelliJ IDEA. Для этого можно использовать команду **Analyze | View Offline Inspection Result** (Анализ | Просмотр результатов инспекции) (*см. раздел 3.2.7.2*).

6.4.4.2. *General (Общие)*

В данную группу входят инспекции, не вошедшие ни в одну из прочих групп.

- **Deprecated symbol** — использован символ (класс, метод), являющийся устаревшим (`@deprecated`).
- **Unused import** — не используемый импорт.
- **Unused symbol** — поле класса или переменная, объявленная, но не используемая.
- **Unused throws declaration** — исключение, объявленное в качестве возможного исключения метода (в списке `throws`), но ни разу не выбрасываемое.
- **Silly assignment** — присваивание переменных самим себе, например, `a = a;`.
- **Access static member via instance reference** — доступ к статическому члену класса (полю или методу) через экземпляр этого класса.

- **Wrong package statement** — определения пакета, которые не соответствуют структуре директорий в проекте.
- **Illegal package dependencies** — зависимости между пакетами, которые настроены как недопустимые. Список недопустимых зависимостей настраивается в диалоге (рис. 6.41), открываемом с помощью кнопки **Configure dependency rules** (Конфигурировать правила зависимостей) в разделе **Options** (Настройки). Более подробно про анализ зависимостей и их конфигурацию можно прочитать в *разделе 6.4.2*.

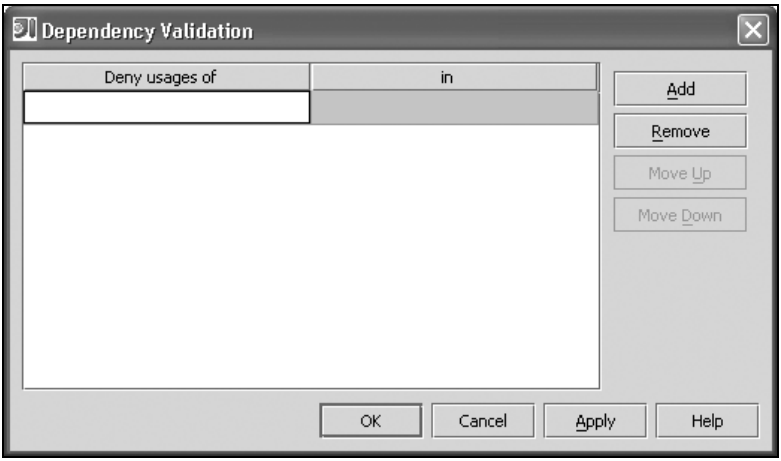


Рис. 6.41. Настройка проверки зависимостей

- **JavaDoc errors** — показывать ошибки в комментариях JavaDoc, например, сломанные ссылки в тегах `@link` и `@see`, неполных параметрах и т. п. Все, кроме неизвестных тегов JavaDoc, которые настраиваются с помощью следующей опции.
- **Unknown javadoc tags** — неизвестные теги JavaDoc. Для данной опции имеется возможность перечислить теги, которые данная проверка не будет считать ошибочными. Такие теги можно ввести в поле **Additional JavaDoc Tags** (дополнительные теги JavaDoc).
- **EJB errors** — определяет нарушения спецификации EJB-уровня ошибок (`error-level`).
- **EJB warnings** — определяет нарушения спецификации EJB-уровня замечаний (`warning-level`).

6.4.4.3. *Local Code Analysis (Локальный анализ кода)*

В эту группу входят следующие инспекции:

- **Constant conditions & exceptions** — данная инспекция находит все условия, значения которых всегда равны `true` или `false`. Также находятся

места в коде, где всегда будут выбрасываться run-time-исключения (RuntimeException). Пример представлен в листинге 6.10.

Листинг 6.10

```
MyManager manager = null;
manager.foo();
```

- **Unused assignment** — неиспользуемое присваивание, при котором значение переменной не применяется после того, как ей было присвоено значение. Причин может быть несколько:
  - переменная никогда не используется после того, как ей было присвоено значение;
  - значение переменной всегда перезаписывается до того, как будет использовано, например `a = 1; a = 2; b = a*a;`
  - переменная никогда не используется;
  - инициализация переменной является излишней по одной из следующих причин (для обоих случаев можно настроить дополнительные опции на панели настроек (**Options**)):
    - ◊ когда оператор `++i` может быть заменен на `(i+1)`;
    - ◊ когда после выполнения оператора `i++` значение переменной `i` не используется;
- **Redundant type cast** — избыточное приведение типа;
- **Local variable or parameter can be final** — локальная переменная или параметр метода может быть объявлен как `final`. Объявлять параметры метода как `final` вообще хорошая практика, которая исключает возможность использования этих параметров в теле функции в качестве локальных переменных, из-за чего бывают очень неприятные ошибки. Для данного типа проверки можно задать две дополнительные опции:
  - **Report local variables** — сообщать о локальных переменных;
  - **Report methods** — сообщать о методах.

6.4.4.4. *Abstraction issues (Проблемы абстракции)*

В эту группу входят следующие инспекции:

- **Cast to a concrete class** — данная инспекция находит все места приведения типов к конкретному классу, а не интерфейсу. Подобные приведения типов могут представлять ошибки в уровне абстракции и могут затруднить тестирование кода. Данная инспекция игнорирует классы, поставляемые в составе Java API, а также в библиотеках сторонних разработчиков;

- **Class reference one of its subclass** — инспекция находит все экземпляры классов, которые содержат ссылки на свои производные классы. Подобные ссылки могут затруднить понимание кода, а также они нарушают некоторые правила объектно-ориентированного дизайна;
- **Collection declared by class, not interface** — инспекция находит все определения переменных типа `Collection`, использующих для определения конкретный класс, а не соответствующий интерфейс.

Следующие пять инспекций работают аналогично — они находят места использования конкретного класса в качестве определения типа, вместо интерфейса. Такое использование может вызвать нарушения абстракции, а также затруднить тестирование. Данные инспекции игнорируют классы, поставляемые в составе Java API, а также в библиотеках сторонних разработчиков.

- **Concrete class for instance variable** — инспекция находит все поля класса.
- **Concrete class for local variable** — инспекция находит локальные переменные. Данная инспекция также игнорирует параметры catch-блоков в виде конкретного класса исключения.
- **Concrete class for method parameter** — инспекция находит все параметры методов.
- **Concrete class for method return** — инспекция находит все возвращаемые значения методов.
- **Concrete class for static variable** — инспекция находит все определения статических переменных.

Далее следуют другие инспекции.

- **Feature Envy** — "Зависть к фитчам" — инспекция, находящая все методы, которые вызывают методы другого класса три раза или более. Вызовы к классам библиотек, базовым классам, внутренним (inner) или внешним (outer) классам со стороны внешних (outer) или внутренних (inner) классов соответственно не учитываются. "Зависть к фитчам" часто показывает, что используемая функциональность находится в неправильном классе и, возможно, должна быть перемещена.
- **'instanceof' a concrete class** — инспекция находит места в коде, в которых проверка с использованием `instanceof` производится конкретным классом, а не интерфейсом. Подобное использование означает излишнюю связность с конкретной имплементацией вместо абстракции. Инспекция не обращает внимания на классы, поставляемые в составе Java API, а также в библиотеках сторонних разработчиков.
- **'instanceof' check for 'this'** — инспекция находит применение оператора `instanceof` для проверки типа `this`. Подобные конструкции означают

ошибку в объектно-ориентированном дизайне и должны быть заменены полиморфными конструкциями.

- **'Magic number'** — инспекция находит все использования "магических чисел" — числовых констант, использованных без определения. "Магические числа" делают код непонятным, а также могут быть причиной появления ошибок, когда "магическое число" меняется в одном месте кода и не меняется в другом. Числа `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `0L`, `1L`, `0.0`, `1.0`, `0.0F` и `1.0F` игнорируются.
- **Public method not exposed in interface** — инспекция находит все `public`-методы класса, которые не представлены в интерфейсе. Представление всех `public`-методов через интерфейсы важно для отслеживания потерь связности и может быть необходимым для хорошего стиля компонентного программирования.
- **'switch' statement** — инспекция находит все применения оператора `switch`. Оператор `switch` часто (но не всегда) является индикатором плохого объектно-ориентированного дизайна.

6.4.4.5. *Class metrics (Метрики класса)*

Данная группа инспекций содержит различные количественные проверки для классов. Для всех инспекций из данной группы можно настроить максимальные количественные границы, на которые они будут реагировать.

- **Class to deep in inheritance tree** — класс находится слишком глубоко в иерархическом дереве. Слишком глубоко наследуемые классы могут затруднять понимание, а также являются хорошим знаком того, что необходим рефакторинг.
- **Class with too many constructors** — класс с большим количеством конструкторов. Классы с большим количеством конструкторов могут приводить к проблемам при инициализации и часто могут быть перестроены с использованием производных классов.
- **Class with too many fields** — класс с большим числом полей. Такие классы часто пытаются "сделать слишком много" и могут нуждаться в рефакторинге в несколько меньших классов. Для данной инспекции можно дополнительно задать возможность учитывать и константные поля.
- **Class with too many methods** — в классе слишком много методов. Причины данной инспекции аналогичны причинам предыдущей.
- **Inner class is too deeply nested** — внутренний класс, слишком "глубоко" размещенный в базовом классе. Размещение внутренних классов внутри внутренних классов — хороший способ запутаться, а также сигнал, что, скорее всего, необходим рефакторинг.

- **Overly complex class** — слишком сложный класс. Данная инспекция высчитывает общую сложность класса как сумму цикломатических (cyclomatic[1]) сложностей всех методов и инициализаций, определенных в классе. Наследуемые методы и инициализации не учитываются.
- **Overly coupled class** — слишком связные классы. Данная инспекция находит все классы, которые имеют слишком много ссылок на другие классы. Такие классы могут быть очень "хрупкими" и легко сломаться. По умолчанию ссылки на системные классы (из пакетов `java` и `javax`) не учитываются, однако эту опцию можно включить. Также имеется опция для включения в инспекцию связности с классами из библиотек.

6.4.4.6. *Class structure (Структура класса)*

Инспекции структуры класса.

- **Abstract class extends concrete class** — абстрактный класс наследуется от конкретного класса.
- **Abstract class without abstract methods** — абстрактный класс без абстрактных методов.
- **Abstract method overrides abstract method** — абстрактный метод переопределяет абстрактный метод.
- **Abstract method overrides concrete method** — абстрактный метод переопределяет конкретный метод.
- **Class may be interface** — класс может быть интерфейсом. Данная инспекция находит абстрактные классы, у которых:
  - нет базового класса (базовый класс `Object`);
  - нет определения полей, кроме `public static final`;
  - нет методов, определенных иначе, чем `public abstract`;
  - нет внутренних классов, которые сами по себе не могут быть интерфейсами.
- **Class name differs from file name** — имя класса отличается от имени файла. Несмотря на то, что Java не запрещает иметь не-`public` классы верхнего уровня, имя которых не совпадает с именем файла, подобная ситуация может ввести в заблуждение, а также снизить применяемость различных утилит для программного обеспечения.
- **Class without constructor** — класс без конструктора. Некоторые стандарты написания программ запрещают такие классы.

---

[1] **http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html**.

- **Class without no-arg constructor** — класс без конструктора по умолчанию (конструктор без аргументов). Такие конструкторы бывают необходимыми, если экземпляры класса создаются с помощью библиотеки reflection.
- **Class without package statement** — класс без описания принадлежности к пакету.
- **Constant declared in abstract class** — константа (поле класса, определенное как `public static final`) определена в абстрактном классе. Некоторые стандарты написания кода требуют, чтобы константы были определены в интерфейсах.
- **Constant declared in interface** — константа определена в интерфейсе. Инспекция, обратная предыдущей. Некоторые стандарты написания кода требуют, чтобы константы были определены в абстрактных классах.
- **Constructor not 'protected' in 'abstract' class** — конструктор абстрактного класса не имеет объявления `protected`.
- **Empty class** — пустой класс — класс, у которого нет полей, методов и конструкторов. Пустые классы часто получаются при больших изменениях или рефакторингах.
- **'final' class** — класс объявлен как `final`. Некоторые стандарты написания кода не рекомендуют использовать `final`-классы.
- **'final' method** — метод объявлен как `final`. Некоторые стандарты написания кода не рекомендуют использовать `final`-методы.
- **'final' method in 'final' class** — метод, объявленный как `final`, находится в классе, объявленном как `final`. Такое описание излишне и может ввести в заблуждение.
- **Inner class of interface** — внутренний класс, описанный внутри интерфейса (`interface`). Некоторые стандарты написания кода не рекомендуют такую конструкцию.
- **Marker interface** — интерфейс без единого метода. Подобные интерфейсы могут ввести в заблуждение и обычно указывают на ошибку дизайна. Интерфейсы, которые наследуют два интерфейса или более, игнорируются.
- **Multiple top level classes** — несколько классов верхнего уровня в одном файле. Размещение нескольких классов верхнего уровня в одном файле может ввести в заблуждение, а также снизить применяемость различных утилит для программного обеспечения.
- **No-op method in abstract class** — пустые методы в абстрактном классе. Обычно бывает лучше, если такие методы определяются как абстрактные (`abstract`) так, чтобы производные классы имели собственные имплементации для таких методов.

- **'private' method declared 'final'** — метод, описанный как `private`, объявлен как `final`. Так как `private`-метод не может быть переопределен, описание `final` для него является излишним.
- **'protected' method in 'final' class** — метод, описанный как `protected`, объявлен как `final`. Такой метод может быть объявлен как `private` или как `package-visible` (видимый для пакета).
- **'public' constructor in non-'public' class** — конструктор, объявленный как `public`, находится в не-`public` классе.
- **Singleton** — инспекция находит все синглтон-классы — классы, определенные таким образом, чтобы иметь лишь один экземпляр. Такие классы усложняют тестирование, и их наличие в программе может свидетельствовать о недочете в объектно-ориентированном дизайне.
- **Static inheritance** — эта инспекция находит наследуемые интерфейсы, которые созданы с единственной целью определения констант. Подобное наследование может ввести в заблуждение и скрыть информацию о важных зависимостях.
- **'static' method declared 'final'** — статический (`static`) метод определен как `final`. Так как статический метод не может быть переопределен, определение его как `final` избыточно.
- **'static', non-'final' field** — данная инспекция находит все статические поля класса, которые не имеют определения `final`. Подобные поля могут быть потенциальной причиной утечек памяти.
- **Utility class** — класс-утилита. Инспекция находит все классы, которые содержат только статические поля и методы. Наличие таких классов может свидетельствовать о недочете в объектно-ориентированном дизайне.
- **Utility class with public constructor** — данная инспекция находит все классы-утилиты (*см. описание предыдущей инспекции*), у которых определен `public`-конструктор. Наличие в классе-утилите `public`-конструктора может ввести в заблуждение, а также может привести к тому, что будут ошибочно созданы экземпляры данного класса.
- **Utility class without private constructor** — класс-утилита без `private`-конструктора. Причины создания данной инспекции аналогичны причинам предыдущей, поскольку если у класса не определен `private`-конструктор, его экземпляр можно создать с помощью конструктора по умолчанию.

6.4.4.7. *Cloning issues (Проблемы клонирования)*

Инспекции, рассматривающие проблемы клонирования.

- **clone() doesn't call super.clone()** — метод `clone()` не содержит вызова `super.clone()`, что может привести к ошибке в инициализации объекта.

- **clone() doesn't declare CloneNotSupportedExeption** — метод `clone()` не имеет определения на выбрасывание исключения `CloneNotSupportedExeption`. Из-за этого производные классы, которые хотят запретить клонирование, не смогут выполнить это стандартным способом. Данная инспекция игнорирует методы `clone()`, определенные как `final`, а также методы `clone()` классов, определенных как `final`.
- **clone() instantiates objects with constructor** — метод `clone()` создают экземпляры объектов с помощью конструкторов. Создание экземпляров класса внутри метода `clone()` должно производиться с помощью вызова метода `clone()` вместо создания объекта напрямую в целях поддержки последующих производных классов.
- **Cloneable class without clone()** — класс, имплементирующий интерфейс `Cloneable`, не содержит переопределения метода `clone()`. Такой класс будет использовать имплементацию метода `clone()` по умолчанию, а такой метод часто может не выполнять требуемых действий.

6.4.4.8. *Code maturity issues (Проблемы устаревшего кода)*

- **Call to prinStackTrace()** — вызов метода `Throwable.printStackTrace()` без параметров. Такой вызов часто используется в целях отладки и, возможно, должен быть удален из готового кода, либо заменен более ясным способом вывода информации.
- **Call to Thread.dumpStack()** — вызов метода `Thread.dumpStack()` часто используется при отладке и, возможно, должен быть удален из готового кода, либо заменен более ясным способом вывода информации.
- **Class without toString()** — класс, у которого не определен метод `toString()`. Удобно иметь для всех классов метод `toString()` для целей отладки.
- **Use of obsolete collection type** — использование устаревших классов коллекций `java.util.Vector` и `java.util.Hashtable`, которые поддерживаются Java, но тем не менее устарели после появления классов коллекций в JDK 1.2 и, возможно, не должны применяться при разработке нового программного обеспечения.
- **Use of System.out or System.err** — `System.out` и `System.err` часто используются при отладке и, возможно, должны быть удалены из готового кода, либо заменены более ясным способом вывода информации.

6.4.4.9. *Code style issues (Стилевые проблемы)*

- **C-style array declaration** — определение массива в стиле C, когда индикатор массива приписывается переменной, а не типу, как это обычно делается в Java (листинг 6.11).

Листинг 6.11

```
int arrayVar [] = new int [10]; // Определение массива в стиле C
int [] arrayVar = new int [10]; // Java-стиль определения массива
```

- **Constant on left side of comparison** — константа в левой части оператора сравнения. Некоторые соглашения о написании кода оговаривают место константы в правой части оператора сравнения.
- **Constant on right side of comparison** — инспекция, противоположная предыдущей.
- **expression.equals("literal") rather than "literal".equals(expression)** — аргумент метода сравнения класса `String` является строковой константой. Некоторые стандарты написания кода рекомендуют применять строковые константы в качестве объекта, для которого вызывается метод `equals()`, а не в качестве аргумента вызова метода. Это позволит снизить вероятность возникновения `NullPointerException`.
- **Missorted modifiers** — инспекция находит все определения, у которых последовательность написания модификаторов отличается от общепринятого порядка.
- **Multiple variables in one declaration** — несколько переменных определено в пределах одного определения. Некоторые стандарты написания кода запрещают такое определение.
- **Return of this** — методы, возвращающие `this`. Несмотря на то, что такой `return` верен с точки зрения языка Java, он редко бывает необходимым и обычно используется в методах, которые можно вызывать по цепочке, вроде того, что представлен в листинге 6.12. Подобные цепочки не приветствуются многими стандартами написания кода.

Листинг 6.12

```
buffer.append("foo").append("bar").append("baz");
```

- **Unnecessarily qualified static usage** — инспекция, находящая вызовы статических полей или методов класса в пределах данного класса, использующих квалификатор имени класса. Такое использование является избыточным и безболезненно может быть удалено.
- **Unqualified static usage** — эта инспекция является обратной предыдущей. Она находит вызовы статических полей или метода класса внутри данного класса, которые не используют квалификатор имени класса. Такое использование может ввести в заблуждение.

- **Variables of different types in one declaration** — инспекция находит все определения переменных нескольких типов в пределах одного определения. Такие определения могут ввести в заблуждение.

6.4.4.10. *Encapsulation issues (Проблемы инкапсуляции)*

- **Assignment to Collection or array field from parameter** — инспекция находит все попытки модифицировать поле коллекции (`Collection`) или массива, переданного в качестве параметра. Содержимое коллекции или массива может быть модифицировано так, что конструкция повлечет неожиданное изменение состояния объекта. Часто такая конструкция используется в целях повышения производительности, однако она является местом потенциальных ошибок.
- **Package visible field** — поле класса, видимое на уровне пакета. Данная инспекция игнорирует константные поля, описанные с помощью модификаторов `static final`.
- **Package visible inner class** — внутренний класс, видимый на уровне пакета.
- **Protected field** — поле класса, имеющее модификатор доступа `protected`. Инспекция игнорирует константные поля, описанные с помощью модификаторов `static final`.
- **Protected inner class** — внутренний класс, имеющий модификатор доступа `protected`.
- **Public field** — поле класса, имеющее модификатор доступа `public`. Инспекция игнорирует константные поля, описанные с помощью модификаторов `static final`.
- **Public inner class** — внутренний класс, имеющий модификатор доступа `public`.
- **Return of Collection or array field** — инспекция находит все возвраты коллекций (`Collection`) и массивов из методов. Содержимое коллекции или массива может быть модифицировано так, что конструкция может повлечь неожиданное изменение состояния объекта. Часто такая конструкция используется в целях повышения производительности, однако она является местом потенциальных ошибок.

6.4.4.11. *Error handling (Проблемы обработки ошибок)*

- **'catch' generic class** — инспекция находит все блоки `catch`, обрабатывающие общие классы исключений. Для точности и ясности кода рекомендуется использовать конкретные типы исключений и ошибок в блоках `catch` вместо общих:
  - `java.lang.Throwable`;

- `java.lang.Exception`;
- `java.lang.RuntimeException`.

- **Checked exception class** — инспекция находит все проверяемые типы исключений (т. е. производные классы `Exception`, которые не являются производными от `RuntimeException`). Некоторые стандарты написания кода требуют, чтобы исключения, определенные пользователем, не проверялись.
- **'continue' or 'break' inside 'finally' block** — оператор `continue` или `break` внутри блока `finally`. Такое использование этих операторов очень сбивает с толку, может скрыть возникшие исключения и существенно усложнить отладку.
- **Empty 'catch' block** — пустой `catch`-блок. Такая конструкция существенно усложняет отладку. Для данной инспекции можно настроить дополнительные опции:
  - **Comments count as content** — комментарии внутри `catch`-блока засчитываются как содержимое (инспекция будет засчитывать такую конструкцию как правильную);
  - **Ignore empty catch blocks in JUnit test cases** — игнорировать пустые `catch`-блоки в тестах JUnit.
- **Empty 'finally' block** — пустой `finally`-блок. Такой блок обычно свидетельствует об ошибке в коде.
- **Empty 'try' block** — пустой `try`-блок.
- **Error not rethrown** — инспекция находит все `catch`-блоки, обрабатывающие ошибку `java.lang.Error` или его производные классы, которые не выбрасывают ошибку дальше. `java.lang.ThreadDeath` игнорируется данной инспекцией.
- **'instanceof' on 'catch' parameter** — инспекция, находящая все проверки типа исключения внутри `catch`-блока. Вместо такой конструкции гораздо лучше использовать несколько `catch`-блоков — каждый для своего исключения.
- **Nested 'try' statement** — вложенные `try`-блоки. Такая конструкция может ввести в заблуждение и, возможно, должна быть объединена с внешним `try`-блоком путем объединения `catch`- и `finally`-блоков.
- **Overly broad 'catch' block** — инспекция находит `catch`-блоки, параметры которых содержат исключения, отличные от тех, что могут быть выброшены внутри `try`-блока.
- **'return' inside 'finally' block** — оператор `return` внутри `finally`-блока. Такая конструкция может скрывать исключения и существенно усложнить отладку.

- **ThreadDeath not rethrown** — инспекция находит все `catch`-блоки, обрабатывающие ошибку `java.lang.ThreadDeath`, которые не выбрасывают ошибку дальше.
- **'throw' generic class** — инспекция находит все места в коде, которые выбрасывают общие классы исключений. Для четкости и ясности рекомендуется выбрасывать конкретные исключения, а не общие:
  - `java.lang.Throwable`;
  - `java.lang.Error`;
  - `java.lang.Exception`;
  - `java.lang.RuntimeException`.
- **'throw' inside 'catch' block which ignores the caught exception** — инспекция находит все исключения, выбрасываемые внутри `catch`-блока, которые не "оборачивает" (wrap) исходное исключение. В таком случае информация о строках из основного исключения будет потеряна, что не очень хорошо.
- **'throw' inside 'finally' block** — выбрасывание исключения внутри `finally` блока. Такая конструкция может скрыть другие исключения, а также существенно затрудняет отладку.
- **Unchecked exception class** — инспекция находит все исключения, которые не обрабатываются `catch`-блоками (например, `java.lang.RuntimeException`). Некоторые стандарты кодирования требуют, чтобы все исключения, определенные пользователем, были обработаны.
- **Unused 'catch' parameter** — инспекция находит все `catch`-блоки, параметр которых не используется внутри них.

6.4.4.12. *Finalization issues (Проблемы финализации)*

- **finalize() called explicitly** — непосредственный вызов `finalize()`. Вызывать `Object.finalize()` непосредственно — плохая идея, которая может привести объекты в противоречивое состояние. Вызовы `super.finalize()` из имплементации `finalize()` игнорируются.
- **finalize() doesn't call super.finalize()** — имплементация метода `finalize()` не вызывает `super.finalize()`. Если забыть вызвать `super.finalize()`, то объект может неправильно очистить свои ресурсы.
- **finalize() not declared 'protected'** — метод `finalize()` не имеет модификатора `protected`. Это должно быть сделано, чтобы предотвратить возможность вызова метода извне.
- **Use of finalize()** — эта инспекция находит все места использования метода `finalize()`. Некоторые стандарты написания кода рекомендуют не использовать этот метод, поскольку нельзя гарантировать, что он будет вызван.

6.4.4.13. *Imports (Проблемы импорта)*

- *** import** — использование оператора `import` с маской '`*`' (импорт пакетов). Некоторые стандарты написания кода запрещают такие операторы импорта.
- **Import from same package** — использование оператора импорта для классов из того же пакета. Такие операторы импорта необязательны и, возможно, являются результатом незавершенного рефакторинга.
- **java.lang import** — импорт классов из пакета `java.lang`. Такие операторы импорта необязательны.
- **Redundant import** — инспекция находит операторы импорта, которые перекрываются предыдущими операторами импорта в том же самом файле.
- **Single class import** — использование оператора `import` для каждого класса в отдельности. Некоторые стандарты написания кода запрещают такие операторы импорта.
- **Static import** — инспекция находит все операторы статического импорта. Данные операторы импорта не поддерживаются Java версии 1.4 и ниже.
- **Unused import** — инспекция находит все неиспользуемые операторы импорта. Данная инспекция дублирует другую инспекцию IntelliJ IDEA из раздела **General** (Общие).

6.4.4.14. *Initialization issues (Проблемы инициализации)*

- **Abstract method call in constructor** — вызовы абстрактного метода в конструкторе абстрактного класса. Такие вызовы могут повлечь ошибки, поскольку нельзя гарантировать, что объект полностью инициализирован до того, как произошел вызов метода.

Следующие четыре инспекции использует очень консервативный алгоритм и могут неправильно определять некоторые поля класса как не проинициализированные. Однако те поля класса, которые распознаются ими как проинициализированные — действительно проинициализированы. У данных инспекций есть опция **Ignore primitive types** (Игнорировать примитивные типы). Порядок следования инспекций несколько изменен. Третья и четвертая инспекции в диалоге настроек расположены после инспекции **Overridable method call in constructor** (Вызов потенциально переопределяемого метода в конструкторе).

- **Instance variable may not be initialized** — инспекция находит все поля класса, которые могли быть не проинициализированы.
- **Instance variable used before initialized** — поле класса используется до своей инициализации.
- **Static variable may not be initialized** — статическая переменная может быть не проинициализирована.

- **Static variable used before initialization** — статическая переменная используется до своей инициализации.

Далее идут следующие инспекции:

- **Overridable method call in constructor** — вызов потенциально переопределяемого метода (метода, у которого нет модификаторов `final`, `static`, `private`) в конструкторе. Подобные вызовы могут привести к ошибкам, поскольку объект может оказаться не полностью проинициализированным в момент вызова метода.
- **'this' reference escaped in constructor** — данная инспекция находит все возможные исключения использования `this` в конструкторе объекта. Исключения составляет применение `this` в качестве параметра вызова метода или объекта присваивания в конструкторе или инициализации. Подобные исключения могут привести к ошибкам, поскольку объект становится доступным в контексте, в котором он, возможно, не был правильно проинициализирован.

6.4.4.15. *Internationalization issues (Проблемы интернационализации)*

Данная группа инспекций определяет места в коде, которые обычно работают неправильно в интернационализированном окружении.

- **Call to Date.toString()** — вызов метода `toString()` для объектов типа `java.lang.Date`.
- **Call to Numeric.toString()** — вызов метода `toString()` для числовых объектов.
- **Call to String.compareTo()** — вызов метода `compareTo()` для строковых объектов.
- **Call to String.equals()** — вызов метода `equals()` для строковых объектов.
- **Call to String.equalsIngoreCase()** — вызов метода `equalsIngoreCase()` для строковых объектов.
- **Call to Time.toString()** — вызов метода `toString()` для объектов типа `java.sql.Time`.
- **Character comparison** — инспекция находит все места сравнения значений типа `char`.
- **Hardcoded string literals** — инспекция находит все строковые выражения, жестко прописанные в коде. Эта инспекция игнорирует:
  - пустые строки;
  - строки, состоящие из пробелов;
  - строки, используемые только в качестве аргумента метода, возвращающего строку.

- **"Magic character"** — инспекция находит все символьные константы, используемые без объявления — "магические символы" (по аналогии с "магическими числами" (*см. раздел 6.4.4.4*)). Использование "магических символов" делает код неясным и может привести к возникновению ошибок, если такой "магический символ" изменяется в одном месте, но не меняется в остальных.
- **String concatenation** — инспекция находит все конкатенации (оператор +) строк (`String`). Конкатенация может быть заменена использованием `java.text.MessageFormat` или сходными классами.
- **Use of StringTokenizer** — использование класса `StringTokenizer`.

6.4.4.16. *JUnit issues (Проблемы тестов JUnit)*

Данные инспекции проверяют правильность написания JUnit-тестов.

- **JUnit test case with not trivial constructor** — инспекция находит все тесты JUnit, имеющие сложную логику в конструкторах. Вместо этого инициализация тестов JUnit должна производиться с помощью метода `setUp()`.
- **Missing message on JUnit assertion** — вызовы метода JUnit `assertXXX()` без параметра с сообщением. Сообщение об ошибке может помочь распознать намерения теста.
- **setUp() doesn't call super.setUp()** — метод `setUp()` из теста JUnit не вызывает `super.setUp()`.
- **setUp() with incorrect signature** — метод `setUp()` имеет неправильную сигнатуру. Он должен иметь модификатор доступа `public`, не иметь возвращаемого параметра (`void`) и не иметь параметров вызова. Такой метод `setUp()` не будет вызван JUnit, соответственно тест не будет правильно проинициализирован.
- **'setup()' instead of 'setUp()'** — ошибка в написании названия метода.
- **'suite()' method not declared static** — метод `suite()` не имеет модификатора `static`.
- **tearDown() doesn't call super.tearDown()** — метод `tearDown()` не вызывает `super.tearDown()`.
- **tearDown() with incorrect signature** — метод `tearDown()` имеет неправильную сигнатуру. Он должен иметь модификатор доступа `public`, не иметь возвращаемого значения (`void`) и не иметь параметров вызова.
- **'teardown()' instead of 'tearDown()'** — ошибка в написании названия метода.
- **Test method with incorrect signature** — инспекция находит все методы класса теста JUnit, имена которых начинаются с '`test`', но которые не имеют модификатора доступа `public`, имеют возвращаемые параметры или принимают какие-либо аргументы.

6.4.4.17. *Method metrics (Метрики методов)*

Данная группа инспекций похожа на инспекции, проверяющие метрики класса (Class metrics). Данная группа содержит различные количественные проверки для методов. Для некоторых инспекций из данной группы можно настроить максимальные количественные границы, на которые они будут реагировать.

- **Method with more than three negotiations** — метод, в котором использовано более чем три оператора отрицания (`!` или `!=`). Такие методы могут быть неясными.
- **Method with multiple loops** — метод с несколькими операторами циклов.
- **Method with multiple return points** — метод с несколькими точками выхода. Такие методы могут быть сложны для понимания, а также плохо поддаваться рефакторингу.
- **Method with too many parameters** — метод с большим числом параметров. Присутствие такого метода — хороший знак, что пора заняться рефакторингом.
- **Overly complex method** — слишком сложный метод. Данная инспекция находит все методы, которые имеют слишком высокую цикломатическую (cyclomatic) сложность. Вкратце, цикломатическая сложность — это количество точек пересечения в методе. Методы с высокой цикломатической сложностью могут быть трудны для понимания и для тестирования.
- **Overly coupled method** — слишком связный метод. Данная инспекция находит все методы, которые сильно связны, т. е. имеют много ссылок на другие классы. Такие методы довольно "хрупкие" и могут быть легко испорчены. Для данной инспекции можно задать дополнительные опции:
  - **Include coupling to java system classes** — учитывать ссылки на системные классы (из пакетов `java.` или `javax`);
  - **Include coupling to library classes** — учитывать ссылки на классы из библиотек.
- **Overly long method** — слишком длинный метод. Такие методы сложны для понимания. Хорошей практикой является их рефакторинг на несколько более коротких методов. Для данной инспекции можно настроить число строк кода (вне комментариев), начиная с которого данная инспекция должна обращать внимание.
- **Overly nested method** — метод с большим количеством вложенных конструкций. Такие методы сложны для понимания и являются хорошим знаком того, что, возможно, пора заняться рефакторингом.

6.4.4.18. *Naming conventions (Соглашения об именах)*

В данной группе можно выделить подгруппу однотипных инспекций, проверяющих имена символов на соответствие регулярному выражению, заданному в дополнительной опции (выражение задается в стандартном для `java.util.regex` формате), а также проверяющих их имена на соответствие минимальной и максимальной длине, заданной дополнительными опциями:

- **Class naming convention** — инспекция проверяет имена классов;
- **Constant naming conversion** — инспекция проверяет имена констант. Константы — это переменные, не зависящие от типа, объявленные с модификаторами `static final`;
- **Instance method naming convention** — инспекция проверяет имена методов. Методы, переопределяющие методы классов из библиотек, не учитываются;
- **Instance variable naming convention** — инспекция проверяет имена полей класса;
- **Interface naming convention** — инспекция проверяет имена интерфейсов;
- **Local Variable naming convention** — инспекция проверяет имена локальных переменных. Для данной инспекции можно задать две дополнительные опции:
  - **Ignore for-loop parameters** — игнорировать параметры циклов `for`;
  - **Ignore catch block parameters** — игнорировать параметры `catch`-блоков;
- **Method parameter naming convention** — инспекция проверяет имена параметров метода;
- **Static method naming convention** — инспекция проверяет имена статических методов;
- **Static variable naming convention** — инспекция проверяет имена статических переменных. Константы, т. е. переменные, объявленные с модификаторами `static`, `final`, не учитываются.

Далее перечислены остальные инспекции данной группы:

- **Class name prefixed with package name** — имя класса начинается с названия пакета (без учета регистра символов). Такие имена иногда встречаются, однако это свидетельствует о плохой схеме именования;
- **Confusing main() method** — инспекция находит все методы `main`, сигнатура которых отличается от `public static void main(String[])`. Такие методы вводят в заблуждение, поскольку метод `main` предполагается как точка входа приложения;

- **Exception class name doesn't end with Exception** — имя класса исключения не оканчивается на 'Exception';
- **Method name same as class name** — название метода совпадает с именем класса. Такое название может ввести в заблуждение;
- **Method name same as parent class name** — название метода совпадает с именем родительского класса. Такое название может ввести в заблуждение;
- **Non-exception class name ends with 'Exception'** — класс, не являющийся исключением (производным от Exception), имеет название, заканчивающееся на 'Exception';
- **Standard variable names** — инспекция находит все переменные со "стандартными" именами и имеющие "неподходящий" тип. Такие имена переменных могут ввести в заблуждение:
  - i, j, k, m, n — тип int;
  - f — тип float;
  - d — тип double;
  - b — тип byte;
  - c, ch — тип char;
  - l — тип long;
  - s, str — тип String;
- **Use of '$' in identifier** — использование символа '$' в составе идентификаторов. Несмотря на то, что это разрешено в языке Java, использование таких идентификаторов за пределами сгенерированного Java-кода строго не рекомендуется.

6.4.4.19. *Performance issues (Проблемы производительности)*

Данная группа инспекций выполняет поиск конструкций кода, которые могут привести к проблемам с производительностью.

- **Boolean constructor call** — создание нового объекта типа Boolean. Вызов конструктора Boolean нужен редко и часто приводит к проблемам с производительностью.
- **Calls to System.gc() or Runtime.gc()** — инспекция находит все вызовы System.gc() и Runtime.gc(). Использование этих вызовов может быть полезным при тестировании, однако использовать их в готовом коде — плохой тон.
- **Collection without initial capacity** — инспекция находит все попытки создать новый объект — коллекцию (Collection) — без указания его на-

чальной вместимости. Когда начальная вместимость не указана, используется вместимость по умолчанию, которая редко бывает оптимальной. Если потом по мере работы с коллекцией ее вместимость будет меняться, это может привести к проблемам с производительностью. Данная инспекция проверяет следующие классы:

- `java.util.ArrayList`;
- `java.util.BitSet`;
- `java.util.HashMap`;
- `java.util.HashSet`;
- `java.util.Hashtable`;
- `java.util.Vector`;
- `java.util.WeakHashMap`.

- **Field may be 'static'** — полю класса может быть добавлен модификатор `static`. Поле может быть статическим, если оно имеет модификатор `final`, и проинициализировано константой.
- **Field repeatedly accessed in method** — инспекция находит методы, которые имеют обращение к некоторому полю класса три раза и более, а также обращение к полю в цикле. Такое обращение может быть логически верным, но с точки зрения производительности, лучше заменить это обращение локальной переменной, временно скопировав в нее поле класса, а также при необходимости скопировав ее значение обратно в поле класса. Для данной инспекции можно задать дополнительную опцию **Ignore final field** — игнорировать поля класса, имеющие модификатор `final`. Дело в том, что многие компиляторы и JVM грамотно оптимизируют обращение к такой переменной и не требуют создания временной локальной переменной.
- **Inner class may be 'static'** — внутренний класс может быть объявлен статическим. Это может быть сделано тогда, когда внутренний класс не имеет обращений к экземпляру основного класса. Сделав его статическим, можно сэкономить память.
- **Manual array copy** — инспекция находит все места в коде, где "вручную" производится копирование массивов, которые могут быть заменены на вызов `System.arraycopy()`. Данная инспекция может пропустить сложные конструкции копирования массивов, так что стоит использовать ее с осторожностью.
- **Method may be 'static'** — метод может быть объявлен статическим. Это возможно, если метод не имеет ссылок на класс, на нестатические методы или поля класса, а также не переопределен в производных классах.

- **Multiply or divide by power of two** — инспекция находит все операции деления и умножения целочисленных значений на 2. Данные выражения могут быть заменены левым или правым сдвигом.
- **Object allocation in loop** — объект создается в цикле. Возможно, это не является проблемой в конкретном случае, однако часто такое создание может приводить к проблемам с производительностью, а также к утечкам памяти.
- **Redundant String constructor call** — излишний вызов конструктора при создании объекта `String` для копирования имеющейся строки. Такое создание объекта `String` редко необходимо и может привести к проблемам с производительностью, если выполняется часто.
- **Redundant String.toString()** — излишний вызов `String.toString()`.
- **Single character .startsWith() or .endsWith()** — инспекция ищет все вызовы методов `String.startsWith()` и `String.endsWith()` со строкой из одного символа в качестве входного параметра. Такой вызов лучше заменить методом `String.charAt()`.
- **Single character string concatenation** — конкатенация строк из одного символа. Такие строки могут быть заменены символьными литералами.
- **Static collection** — коллекция объявлена статической. Может быть, это не является проблемой, однако часто такая коллекция служит причиной утечек памяти, так что некоторые стандарты написания кода запрещают статические коллекции.
- **String concatenation in loop** — конкатенация строк в цикле. Почти всегда такую конкатенацию лучше заменить более ясным вызовом метода `StringBuffer.append()`.
- **StringBuffer or StringBuilder without initial capacity** — создание объекта `StringBuffer` или `StringBuilder` без указания начальной вместимости. Когда начальная вместимость не указана, используется вместимость по умолчанию, которая редко бывает оптимальной. Если потом по мере работы с объектом `StringBuffer` его вместимость будет меняться, это может привести к проблемам с производительностью.
- **StringBuffer.toString() in concatenation** — использование значения `StringBuffer.toString()` при конкатенации строк. Мало того, что это может ввести в заблуждение, так еще и приводит к созданию новой строки и ее копированию, и является излишним в JDK 1.4.
- **Tail recursion** — рекурсия, при которой последним оператором метода перед его завершением является вызов того же самого метода. Такая рекурсия всегда может быть заменена циклом, который будет выполняться значительно быстрее. Некоторые JVM самостоятельно производят такую оптимизацию, но не все, так что использование такой рекурсии может дать разную производительность на разных JVM.

- **Unnecessary temporary object in conversion from String** — данная инспекция находит все нежелательные конструкции создания экземпляров объектов при конвертировании из строк (`String`) в простые типы. Например, выражение `new Integer("3").intValue()` может быть заменено выражением `Integer.valueOf("3")`.
- **Unnecessary temporary object in conversion to String** — данная инспекция находит все нежелательные конструкции создания экземпляров объектов простых типов при конвертировании их в строки (`String`). Например, выражение `new Integer(3).toString()` может быть заменено выражением `Integer.toString(3)`.
- **Use of java.lang.reflect** — использование любых классов из пакета `java.lang.reflect`. Несмотря на свои большие возможности, рефлексия в Java работает достаточно медленно, а также может быть небезопасной с точки зрения отсутствия контроля типов и исключений на этапе компиляции.
- **Zero-length array allocation** — создание объектов нулевой длины. Так как в Java размер массивов — величина постоянная, в большинстве случаев бывает лучше иметь предопределенные пустые массивы, используемые из разных мест кода, чем каждый раз создавать такой массив заново. Данная инспекция игнорирует пустые массивы, созданные как поля класса, и массивы, имеющие модификаторы `static`, `final`.

6.4.4.20. *Portability issues (Проблемы переносимости)*

- **Annotation** — инспекция находит использования аннотаций. Аннотации не поддерживаются JDK версии 1.4 и ниже.
- **Annotation class** — инспекция находит использования аннотаций интерфейсов. Аннотации интерфейсов не поддерживаются JDK версии 1.4 и ниже.
- **'assert' statement** — оператор `assert`. Данный оператор не поддерживается JDK версии 1.3 и ниже.
- **Auto-boxing** — инспекция находит использования "auto-boxing" — автоматического обертывания (wrapping) значений простых типов в объекты. "Auto-boxing" не поддерживается JDK версии 1.4 и ниже.
- **Auto-unboxing** — инспекция, определяющая обратное преобразование "auto-unboxing" значений объектов в значения простых типов. "Auto-unboxing" не поддерживается JDK 1.4 и ниже.
- **Call to Runtime.exec()** — вызов различных вариантов метода `Runtime.exec()`. Данные вызовы наследственно непереносимы.
- **Call to System.exit()** — вызов метода `System.exit()`. Использование такого вызова делает код непереносимым на многие серверы приложений.

- **Call to System.getenv()** — вызов метода `System.getenv()`. Данные вызовы наследственно непереносимы.
- **Enumerated class** — использование классов-перечислений `enum`. Данная конструкция не поддерживается JDK версии 1.4 и ниже.
- **Extended 'for' statement** — расширенный синтаксис оператора цикла `for` в версии языка Java 5. Такое использование соответственно не поддерживается старыми версиями JDK — 1.4 и ниже.
- **Hardcoded file separator** — инспекция находит все места использования символов прямой '/' и обратной '\' косой черты в строках или символьных литералах. Эти символы обычно используются в качестве разделителя пути файла, так что переносимость может существенно пострадать, если они записаны прямо в строке. Данная инспекция игнорирует прямую косую черту '/' сразу после символа двойной кавычки '"', или непосредственно перед символом '>', т. к. такие конструкции часто встречается в XML- и HTML-тегах.
- **Hardcoded line separator** — данная инспекция находит все места использования символов новой строки '\n' и возврата каретки '\r' в строках и символьных литералах. Эти символы часто используются в качестве обозначения конца строки, так что переносимость может существенно пострадать, если они записаны прямо в строке.
- **Native method** — инспекция находит все методы, определенные с модификатором `native`. Такие методы непереносимы.
- **Use of 'assert' as identifier** — инспекция находит все переменные, методы и классы, имеющие название '`assert`'. Такие имена доступны в JDK версии 1.3 и ниже, однако в более поздних версиях JDK '`assert`' является ключевым словом.
- **Use of 'enum' as identifier** — инспекция находит все переменные, методы и классы, имеющие название '`enum`'. Такие имена доступны в JDK версии 1.4 и ниже, однако в версии языка Java 5 '`enum`' является ключевым словом.
- **Variable argument method** — использование методов с переменным числом параметров. Такие методы не поддерживаются в JDK версии 1.4 и ниже.

6.4.4.21. *Potentially confusing code constructs (Потенциальные проблемные конструкции кода)*

- **Assignment to catch block parameter** — присваивание параметру `catch`-блока.
- **Assignment to for loop parameter** — присваивание параметру цикла. Данная инспекция находит все присваивания значения переменной в теле цикла `for`. Также учитываются все попытки инкремента и декремента переменной.

- **Assignment to method parameter** — присваивание параметру метода. Также находятся все попытки инкремента и декремента параметра.
- **'break' statement** — инспекция находит все использования оператора `break`, встречающиеся в любых местах за исключением конца веток оператора `switch`. Такое использование усложняет рефакторинг и может ввести в заблуждение.
- **'break' statement with label** — оператор `break` с меткой. Использование таких конструкций усложняет рефакторинг и может ввести в заблуждение.
- **Chained equality comparison** — инспекция находит все цепочные сравнения (т. е. `a==b==c`).
- **Chained method calls** — вызов методов по цепочке, когда объект, для которого вызывается метод, был сам получен в результате вызова метода (т. е. `o.f1().f2()`).
- **Class escapes defined scope** — инспекция находит все ссылки на классы, позволяющие использовать имя класса за пределами области видимости класса. Например, метод класса с модификатором доступа `public`, который возвращает в качестве результата `private` внутренний класс. Или поле с модификатором доступа `protected`, типом которого является класс, доступный только внутри пакета. Несмотря на то, что Java позволяет создавать такие конструкции, они могут ввести в заблуждение и усложняют возможность повторного использования кода.
- **Conditional expression (?:)** — инспекция находит все использования условного оператора. Некоторые стандарты написания кода запрещают использование условного оператора и рекомендуют вместо него применять оператор `if-else`.
- **Confusing else branch** — неясное использование ветки `else` условного оператора `if`. Данная инспекция находит все места в коде, в которых использование оператора `else` в ветвлении `if-else` излишне, например, как в листинге 6.13.

Листинг 6.13

```
String test;
if (args.length == 0) {
    return;
} else {
    test = args[0];
}
System.out.println("test = " + test);
```

В данном примере использование оператора `else` излишне.

- **Confusing floating point literal** — данная инспекция находит все числа с плавающей точкой, которые записаны:
  - без плавающей точки;
  - без чисел после плавающей точки;
  - без чисел перед плавающей точкой.
- **'continue' statement** — использование оператора `continue`. Его использование может усложнить рефакторинг и быть неясным.
- **'continue' statement with label** — использование оператора `continue` с меткой. Такая конструкция усложняет рефакторинг и может затруднить понимание программы.
- **If statement with negated condition** — данная инспекция находит все операторы `if` с условием, заданным отрицанием, а также имеющие ветку `else`. Если поменять местами ветки этого оператора и убрать отрицание из условия, конструкция будет проще. Для данной инспекции можно задать опцию **Ignore '!=null' comparison** (Игнорировать сравнения '`!=null`').
- **If statement with too many branches** — оператор `if` с большим количеством скобок. Такая конструкция может ввести в заблуждение, а также свидетельствует о плохом уровне абстракции проектирования. Для данной инспекции можно задать допустимый порог количества скобок.
- **Implicit call to super()** — инспекция находит все конструкторы, которые не начинаются с вызова конструктора родительского класса `super()` либо с вызова конструктора того же самого класса. Такие конструкторы должны начинаться с вызова `super()`. Некоторые стандарты кодирования требуют, чтобы конструкторы начинались с вызова `super()`.
- **Implicit numeric conversations** — автоматическое преобразование чисел. Такое преобразование — само по себе не проблема, однако может быть затруднить трассирование ошибок.
- **Local variable used and defined in different 'switch' branches** — инспекция находит все переменные, определенные в одной ветке оператора `switch` и используемые в другой. Такая конструкция может ввести в заблуждение.
- **Long literal ending with 'l' not 'L'** — инспекция находит все длинные (`long`) числа, записанные со строчной буквы '`l`', а не с заглавной '`L`'. В такой записи букву '`l`' можно легко принять за число `1` (единица).
- **Method names differing only by case** — названия методов отличаются только регистром символов.
- **Nested 'switch' statement** — вложенный оператор `switch`.
- **Nested assignment** — вложенное присваивание. Присваивание, расположенное внутри других операторов, может легко запутать, а также оно

нарушает главный принцип дизайна — каждая конструкция должна выполнять только одно действие.

- **Nested conditional expression** — вложенное выражение сравнения.
- **Nested method calls** — вызов метода используется в качестве параметра для вызова другого метода.
- **Numeric cast that loses precision** — преобразования между простыми числовыми типами, в которых может быть потеряна дробная часть. Сами по себе такие преобразования не являются проблемой, однако они могут усложнить отладку и поиск ошибок.
- **Overloaded methods with same number of parameters** — перегруженные методы одного класса с одинаковым числом параметров.
- **Overly complex arithmetic expression** — слишком сложное арифметическое выражение с большим числом операндов. Для данной инспекции можно настроить порог реагирования.
- **Overly complex boolean expression** — слишком сложное булевское выражение с большим числом операндов. Для данной инспекции также можно настроить порог реагирования.
- **Refused bequest**[1] — инспекция находит все методы класса, которые переопределяют другие методы и не содержат в начале вызова базового метода с помощью `super` из родительского класса. Такие методы могут свидетельствовать о нарушении в абстракции и привести к ошибкам, которые очень сложно обнаружить. Методы, переопределяющие методы из `java.lang.Object`, не учитываются.
- **Switch statement with too few branches** — оператор `switch` с небольшим числом веток. Такие операторы лучше заменять оператором `if`. Для данной инспекции можно настроить порог минимально допустимого числа `case` веток.
- **Switch statement with too many branches** — оператор `switch` со слишком большим числом веток. Данная инспекция также позволяет настроить порог максимально допустимого числа веток.
- **Value of ++ or — used** — инспекция находит все операторы инкремента и декремента, встречающиеся в составе других выражений. Подобные выражения могут быть неясными и к тому же нарушают основной принцип дизайна — одна конструкция должна выполнять только одно действие.

[1] Это не опечатка в слове "request". Здесь действительно имеется в виду слово "bequest" — англ. "акт завещания", "наследство".

6.4.4.22. *Probable bugs (Возможные ошибки)*

- **Assignment to 'null'** — переменной присваивается значение `null` за пределами ее объявления. Такую конструкцию иногда используют, чтобы система сборки мусора могла очистить занимаемую этой переменной память. Однако это увеличивает вероятность возникновения `NullPointerException` и часто свидетельствует о том, что разработчик не совсем понимает семантику класса.
- **Assignment used as condition** — присваивание, используемое в качестве условия в операторах `if`, `while`, `for` или `do`. Такая конструкция способна ввести в заблуждение и может свидетельствовать об опечатке (= вместо ==).
- **'compareto()' instead of 'compareTo()'** — инспекция находит все методы `compareto()`, принимающие один параметр. Обычно это опечатка названия `compareTo()`.
- **Comparison of short and char values** — сравнение значений типов `short` и `char`. Такое сравнение может повлечь ошибки из-за того, что `short` — это знаковый тип, а `char` — незнаковый.
- **Covariant compareTo()** — инспекция находит классы, у которых нет метода `compareTo()`, принимающего в качестве аргумента `java.lang.Object`, но есть другие методы `compareTo()`, принимающие другие типы параметров. Обычно это свидетельствует об ошибке.
- **Covariant equals()** — инспекция, аналогичная предыдущей, только для метода `equals()`.
- **'default' not last case in 'switch'** — ветка `default` не является последней в операторе `switch`, т. е. после нее следуют другие ветки `case`.
- **equals() between objects of inconvertible types** — сравнение "несравнимых" объектов с помощью вызова метода `equals`. Несмотря на то, что такой вызов теоретически может быть использован, в большинстве случаев он свидетельствует об ошибке.
- **Fallthrough in 'switch' statement** — "проваливание" в операторе `switch`. Пример "проваливания" приведен в листинге 6.14. Такая конструкция сложна для понимания, а также часто бывает результатом опечатки (пропущен оператор `break` в конце первой ветки `case`).

Листинг 6.14

```
switch (var) {
    case 1:
        someMethod();
    case 2:
```

```
        anotherMethod();
        break;
}
```

- **Floating point equality comparison** — сравнение чисел с плавающей точкой с помощью операторов сравнения == и !=. Так как числа с плавающей точкой являются неточными, подобное сравнение может привести к неожиданным результатам.
- **For loop where update or condition doesn't use loop variable** — цикл for, в котором выражения изменения и условия не используют переменную цикла.
- **For loop with missing components** — цикл for с пропущенной частью инициализации, условия или изменения. Некоторые стандарты кодирования запрещают такое использование.
- **'hashcode()' instead of 'hashCode()'** — метод с именем hashcode(), не принимающий параметров. Обычно, это опечатка в названии метода hashCode().
- **Infinity loop statement** — бесконечные циклы for, while и do, которые могут завершиться только при возникновении исключения. Несмотря на то, что такие циклы могут быть корректными, часто они свидетельствуют об ошибках в коде.
- **Infinity recursion** — рекурсивные методы, которые могут завершиться только при возникновении исключения. Такие методы не имеют нормального завершения.
- **Integer division in floating point context** — инспекция находит все выражения с делением целых чисел, результат которых явно или не явно используется как число с плавающей точкой. Такие выражения в большинстве случаев являются ошибкой и могут привести к непредсказуемым результатам.
- **Loop statement that doesn't loop** — операторы цикла for, while и do, тело которых будет выполнено только один раз. Обычно это свидетельствует об ошибке.
- **Non-final field referenced in 'compareTo()'** — данная инспекция находит все имплементации метода compareTo(), который использует поля класса, не имеющие модификатора final. Подобное использование может дать разный результат метода в различных точках времени жизни объекта, что может привести к проблемам при использовании объекта со стандартными классами коллекций.
- **Non-final fields referenced in 'equals()'** — инспекция для метода equals(), аналогичная предыдущей.

- **Non-final fields referenced in 'hashCode()'** — инспекция для метода `hashCode()`, аналогичная двум предыдущим.
- **Object comparison using ==, instead of '.equals()'** — сравнение объектов с использованием оператора `==` вместо метода `equals()`. Данная инспекция не обращает внимания на сравнения строк (`String`), а также на сравнение значения объекта с `null`.
- **Octal and decimal integers in same array** — инициализация массива десятичными и восьмеричными числами одновременно. Такая инициализация бывает при копировании списка чисел, притом, что часть из них может иметь вначале нули, так что такие числа будут интерпретированы Java как восьмеричные.
- **Result of InputStream.read() ignored** — инспекция находит все вызовы метода `read()` класса `java.io.InputStream`, возвращаемое значение которого игнорируется. В возвращаемом значении этого метода содержится количество прочитанных байтов. Так что игнорирование этого значения может привести к возникновению ошибок, которые очень сложно найти.
- **Return of 'null'** — инспекция находит все возвраты значения `null`. Такая конструкция может увеличить вероятность возникновения `NullPointerException`, а также свидетельствует о том, что разработчик не до конца понимает семантику класса. Данная инспекция имеет две дополнительные опции:
  - **Methods that returns arrays** — учитывать методы, возвращающие массивы;
  - **Methods that returns objects** — учитывать методы, возвращающие объекты.
- **Statement with empty body** — операторы `if`, `for`, `while` или `do` с пустым телом. Часто такая конструкция свидетельствует об ошибке. Данная инспекция имеет дополнительную опцию **Include statement bodies that are empty code blocks** — учитывать тела операторов, состоящие из пустых блоков кода (`{}`).
- **String comparison using ==, instead of '.equals()'** — сравнение строк с использованием оператора `==` вместо вызова метода `equals()`.
- **Subtraction in compareTo()** — использование вычитания в методе `compareTo()`. Такое использование может привести к непредсказуемым результатам и трудно обнаруживаемым ошибкам в случае целочисленного переполнения. Сравнение целых чисел напрямую и возврат одного из значений `-1`, `0` или `1` — намного лучше.
- **'switch' statement without 'default' branch** — оператор `switch` без ветки по умолчанию `default`.

- **Text label in 'switch' statement** — инспекция находит все использования меток внутри оператора `switch`. Часто такие конструкции являются результатом ошибки (листинг 6.15).

Листинг 6.15

```
switch (x) {
    case 1:
    case2:   // ошибка!
    case 3:
        break;
}
```

6.4.4.23. *Security issues (Проблемы защиты и безопасности)*

- **Cloneable class in secure context** — инспекция находит все классы, которые могут быть клонированы, т. е. такие классы, которые поддерживают интерфейс `Clonneable`, и их метод `clone()` не определен так, чтобы сразу же выбрасывать ошибку. Использование классов, поддерживающих клонирование, может повлечь проблемы с безопасностью.
- **Deserializable class in secure context** — инспекция находит все классы, которые могут быть десериализованы. Класс может быть десериализован, если он поддерживает интерфейс `Serializable`, и его метод `readObject()` не определен так, чтобы сразу же выбрасывать ошибку. Использование классов, которые можно десериализовать, способно повлечь проблемы с безопасностью.
- **Non-static inner class in secure context** — инспекция находит все внутренние классы, определенные без модификатора `static`. Компиляция такого класса создает скрытые методы родительского класса, видимые на уровне пакета. Это может повлечь проблемы с безопасностью.
- **Serializable class in secure context** — инспекция находит все классы, которые могут быть сериализованы. Класс может быть сериализован, если он поддерживает интерфейс `Serializable`, и его метод `writeObject()` не определен так, чтобы сразу же выбрасывать ошибку. Использование классов, которые можно сериализовать, способно повлечь проблемы с безопасностью.

6.4.4.24. *Serialization issues (Проблемы сериализации)*

- **Instance variable may not be initialized by readObject** — поле класса может быть не проинициализировано, когда класс десериализуется методом `readObject()`. Данная инспекция использует очень консервативный алго-

ритм и может неправильно определять некоторые поля класса как не проинициализированные. Однако те поля класса, которые распознаются как проинициализированные — действительно проинициализированы.

- **Non-serializable class with 'readObject()' or 'writeObject()'** — классы, которые не поддерживают интерфейсы `Serializable` или `Externalizable`, но имеют методы `readObject()` и `writeObject()`. Такие классы обычно свидетельствуют об ошибке программиста.
- **Non-serializable class with serialVersionUID** — классы, которые не поддерживают интерфейсы `Serializable` или `Externalizable`, но имеют поле `serialVersionUID`. Такие классы обычно свидетельствуют об ошибке программиста.
- **'readObject()' or 'writeObject()' not declared 'private'** — методы класса, поддерживающего интерфейс `Serializable` или `Externalizable`, `readObject()` и `writeObject()` не объявлены как `private`. Нет причин, чтобы эти методы имели другую область видимости, чем `private`.
- **'readResolve()' or 'writeReplace()' not declared 'protected'** — методы класса, поддерживающего интерфейс `Serializable` или `Externalizable`, `readOResolve()` и `writeReplace()` не объявлены как `protected`.
- **Serializable class with unconstructable ancestor** — инспекция находит сериализуемые классы, ближайший предок которых не имеет конструктор без параметров. Такие классы не могут быть десериализованы.
- **Serializable class without 'readObject()' and 'writeObject()'** — сериализуемый класс, у которого не определены методы `readObject()` и `writeObject()`. В таком случае при сериализации будут использованы методы сериализации по умолчанию, которые могут быть неоптимальными с точки зрения производительности и переносимости. Данная инспекция имеет опцию **Ignore classes serializable due to inheritance** — игнорировать классы, сериализуемые по причине наследования от сериализуемых классов.
- **Serializable class without serialVersionUID** — сериализуемый класс, у которого не определено поле `serialVersionUID`. При каждом изменении такого класса его прошлые сериализованные версии будут нечитаемыми. Данная инспекция имеет опцию **Ignore classes serializable due to inheritance** — игнорировать классы, сериализуемые по причине наследования от сериализуемых классов.
- **'serialVersionUID' field not declared 'static final'** — инспекция находит все сериализуемые классы, имеющие поле `serialVersionUID`, у которого нет модификаторов `static final`.
- **Transient field in non-serializable class** — несериализуемый класс, у которого есть поля, объявленные как `transient`.

6.4.4.25. *Threading issues (Проблемы многопоточности)*

- **Busy wait** — инспекция находит все вызовы `java.lang.Thread.sleep()` внутри циклов. Подобные вызовы означают состояние активного ожидания ("busy-waiting"). Такая конструкция часто бывает неэффективной и может приводить к "мертвой блокировке" (deadlock) в случае, если поток никогда не снимет блокировку с ресурсов.
- **Call to notify() instead of notifyAll()** — инспекция находит все вызовы `notify()`. В большинстве случаев лучше вызывать `notifyAll()`. См. "Concurrent Java Programming" Doug Lea[1].
- **Call to Thread.run()** — инспекция находит все вызовы метода `run()` для класса `java.lang.Thread` или для любого его производного класса. В большинстве случаев такой вызов свидетельствует об ошибке: скорее всего, вместо него предполагалось вызвать `Thread.start()`.
- **Double-checked locking** — данная инспекция находит все конструкции двойной проверки при синхронизации (double-checked locking). Дополнительную информацию о том, чем плоха двойная проверка и почему она небезопасна, см. **http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html**.
- **Empty 'synchronized' statement** — пустой оператор синхронизации. Несмотря на то, что такая конструкция допустима языком Java, она может ввести в заблуждение и часто свидетельствует об опечатке.
- **Field accessed in both synchronized and unsynchronized context** — инспекция находит все поля, не имеющие модификатора `final`, к которым осуществляется обращение из обоих контекстов: синхронизированного и несинхронизированного. Обращение из конструкторов при инициализации игнорируется. Подобное "частично синхронизированное" обращение бывает из-за недосмотра и может приводить к несовместимым структурам данных.
- **Nested 'synchronized' statement** — вложенные операторы синхронизации бесполезны (если объекты синхронизации в обоих блоках совпадают) либо могут приводить к "мертвым блокировкам" (deadlock).
- **Non-private field accessed in synchronized context** — данная инспекция находит все поля класса, не имеющие модификаторов `private` или `final`, обращения к которым производятся из синхронизированного контекста. Для поля, не имеющего модификатора доступа `private`, нельзя гарантировать, что оно всегда будет доступно только из синхронизированного блока, т. е. возможна проблема "частичной синхронизации" (см. описа-

[1] Ссылка на данную книгу приводится в описании к инспекции, однако мы не вынесли ее в общий список ссылок, поскольку сами ее не читали.

ние инспекции **Field accessed in both synchronized and unsynchronized context** в этом разделе). Обращения из конструкторов при инициализации игнорируется.

- **Non-synchronized method overrides synchronized method** — несинхронизированный метод переопределяет синхронизированный.
- **Synchronization on 'this'** — инспекция находит все блоки `synchronize`, параметром синхронизации которых является `this`. Такие блоки похожи на синхронизированные методы, так что бывает очень сложно отследить, чем заблокирован тот или иной объект. И становится возможным устраивать объектам атаки "отказ в доступе" (denial of service).
- **Synchronization on a non-final field** — синхронизация по полю класса, которое не имеет модификатора `final`. Такая конструкция может приводить к тому, что разные потоки могут быть заблокированы разными объектами, тогда как оперируют с одним объектом. Инспекция имеет дополнительную опцию **Include native methods** — учитывать `native`-методы.
- **'synchronized' method** — инспекция находит все синхронизированные методы. Некоторые стандарты написания кода запрещают применение таких методов и требуют вместо них использовать блоки синхронизации.
- **'wait()' not in loop** — инспекция находит все вызовы метода `wait()`, сделанные вне цикла. Обычно `wait()` используется для приостановки потока до того, как не будет выполнено некоторое условие. Это условие должно проверяться после того, как метод `wait()` завершится. Цикл — самый простой способ, чтобы это сделать.

6.4.4.26. *Verbose or redundant code constructs (Громоздкие или излишние конструкции кода)*

- **Assignment replaceable with operator assignment** — инспекция находит все операторы присваивания, которые могут быть заменены присваивающим оператором. Например, код из листинга 6.16 может быть заменен выражением `i+=5`.

Листинг 6.16

```
int i = 0;
i = i + 5;
```

- **Class explicitly extends java.lang.Object** — наследование класса напрямую определено от `java.lang.Object`.
- **'for' loop may be replaced with 'while' loop** — инспекция находит все циклы `for`, у которых нет операции инициализации и обновления и которые могут быть заменены циклом `while`.

- **'for' loop replaced by 'for each' (J2SDK 5.0 only)** — инспекция находит циклы `for`, которые итерируют по коллекциям или массивам и могут быть заменены с использованием нового синтаксиса оператора `for` языка Java 5. Для безопасности, данная инспекция неактивна, если проект не поддерживает возможности языка Java 5.
- **Pointless arithmetic expression** — инспекция находит бессмысленные арифметические выражения. Такие выражения могут быть результатом автоматического рефакторинга. В любом случае, навряд ли это то, что программист действительно хотел написать. Например:
  - сложение с 0;
  - вычитание 0;
  - умножение на 0 или 1;
  - деление на 1;
  - сдвиг на 0.
- **Redundant Boolean comparison** — инспекция находит все излишние сравнения (`==` или `!=`) с булевыми значениями `true` или `false`. Такие операторы слишком громоздки и могут быть безопасно упрощены.
- **Redundant field initialization** — лишняя инициализация переменных теми значениями, которыми их по умолчанию проинициализировала бы JVM.
- **Unnecessary 'default'** — излишний оператор `default` в операторе `switch`. `default` может быть излишним для оператора `switch`, обрабатывающего перечислимый тип, если он имеет `case`-ветки для всех возможных значений из перечисления.
- **Unnecessary 'final' for method parameter** — инспекция находит все методы, параметры которых определены с помощью модификатора `final`. Некоторые стандарты написания кода не рекомендуют использовать такой модификатор, чтобы иметь более простой и наглядный список параметров.
- **Unnecessary 'if' statement** — инспекция находит все операторы `if`, которые могут быть упрощены до простого присваивания или возврата значения (листинг 6.17).

Листинг 6.17

```
if(foo()) {
    return true;
} else {
    return false;
}
// может быть упрощен:
return foo();
```

- **Unnecessary 'return' statement** — излишний оператор `return` в функции, не возвращающей значение (`void`).
- **Unnecessary 'super()' statement** — вызов конструктора родительского класса без аргументов в самом начале конструктора класса. Данный вызов излишен.
- **Unnecessary 'this' qualifier** — излишнее использование спецификатора `this` для обращения к полям и методам класса. Такое использование вполне может быть сделано автоматическим рефакторингом кода.
- **Unnecessary boxing (J2SDK 5.0 only)** — излишнее обертывание простых типов в соответствующие им объекты. В Java, начиная с версии 5, этого делать не нужно, благодаря поддержке "auto-boxing". Данная инспекция выключена для проектов, не поддерживающих возможности языка Java 5.
- **Unnecessary code block** — инспекция находит блоки кода, которые могут быть легко заменены своим содержимым. Тела операторов `if`, `for`, `while`, `do` в виде блоков кода не учитываются.
- **Unnecessary fully qualified name** — имя класса, записанное с приведением его пакета, может быть сокращено. Данная инспекция имеет дополнительную опцию **Ignore fully qualified names in javadoc comments** — игнорировать полные имена в комментариях JavaDoc.
- **Unnecessary interface modifier** — инспекция находит все излишние модификаторы при определении интерфейса или его компонент.
- **Unnecessary parentheses** — инспекция находит все лишние круглые скобки.
- **Unnecessary semicolon** — инспекция находит все лишние точки с запятой, расположенные между компонентами класса, внутри блоков операторов или после определения класса. Несмотря на то, что язык Java не запрещает такие точки с запятой, они не нужны, и могут быть удалены.
- **Unnecessary unboxing (J2SDK 5.0 only)** — излишнее преобразование объектов в простые типы. В Java, начиная с версии 5, этого делать не нужно, благодаря поддержке "auto-unboxing". Данная инспекция выключена для проектов, не поддерживающих язык Java 5.
- **Unused label** — неиспользованные метки.

6.4.4.27. *Visibility issues (Проблемы видимости)*

- **Field name hides field in superclass** — инспекция находит все поля класса, имеющие такое же имя, что и поля из суперкласса. Такие названия полей вводят в заблуждение и могут быть причиной ошибок. Данная инспекция имеет дополнительную опцию **Ignore superclass fields not visible from subclass** — игнорировать поля суперкласса, невидимые из производного класса.

- **Inner field hides outer class field** — поля внутреннего класса имеют такое же название, что и поля из класса, содержащего данный внутренний класс. Подобные имена могут вводить в заблуждение.
- **Local variable hides member variable** — локальная переменная имеет точно такое же имя, что и видимое поле класса, где используется эта локальная переменная. Данная инспекция имеет две дополнительные опции:
  - **Ignore superclass fields not visible from subclass** — игнорировать поля суперкласса, невидимые из производного класса;
  - **Ignore local variables in static methods** — игнорировать локальные переменные в статических методах.
- **Method overloads method of superclass** — инспекция находит методы, которые имеют то же имя, что и методы из суперкласса, но разные, хотя и совместимые параметры. В данном случае метод из производного класса перегружает (overload) метод из базового класса вместо того, чтобы переопределить (override) его. Если подобная ошибка действительно имела место, она может быть причиной скрытых ошибок.
- **Method overrides private method of superclass** — данная инспекция находит методы, имеющие то же имя, что и методы с модификатором доступа `private` из суперкласса. Такие методы могут привести к неясной семантике. Обычно такое бывает, если `private`-метод из базового класса когда-то имел менее жесткий режим доступа.
- **Method overrides static method of superclass** — метод класса имеет то же имя, что и статический метод из базового класса.
- **Parameter hides member variable** — параметр метода имеет то же самое имя, что и поле класса, в котором определен метод. Данная инспекция имеет следующие дополнительные опции:
  - **Ignore for property setter** — игнорировать параметры функций установки свойств (setter);
  - **Ignore for constructors** — игнорировать параметры конструкторов;
  - **Ignore superclass fields not visible from subclass** — игнорировать поля суперкласса, невидимые из производного класса;
  - **Ignore for static method parameters hiding instance fields** — игнорировать параметры статических методов, скрывающие поля класса.

6.4.2. Анализ зависимостей

Анализ зависимостей можно проводить либо для всего проекта, либо для указанного файла или модуля (*см. раздел 3.2.7.3*). Результат анализа зависимостей представляется в инструментальном окне **Dependency Viewer** (Просмотр зависимостей) (рис. 6.42). Подробнее об этом инструментальном окне рассказано в *разделе 3.4.8.13*.

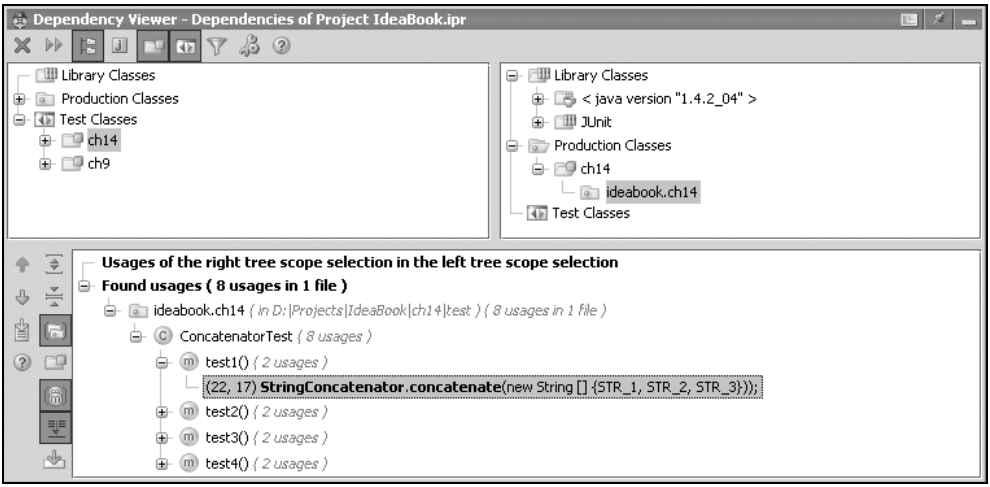


Рис. 6.42. Просмотр результатов анализа зависимостей

Анализ зависимостей между различными модулями и классами может быть полезным, если требуется выделить отдельный модуль в отдельную библиотеку, т. е. если нужно убедиться в том, что модуль содержит все необходимые классы и не имеет ссылок куда-то вовне.

Работая в IntelliJ IDEA, очень легко сделать кучу перекрестных ссылок между классами в разных пакетах и модулях, поскольку IntelliJ IDEA расставляет строки импорта автоматически. В большинстве случае это удобно. Однако, если требуется избежать нежелательных зависимостей, их можно периодически отслеживать с помощью команды анализа зависимостей.

В инструментальном окне анализа зависимостей есть возможность задать список запрещенных зависимостей. Делается это с помощью диалога, который открывается командой **Edit Rules** (Редактировать правила) в верхней панели инструментов окна (рис. 6.43).

Рис. 6.43. Настройка правил проверки зависимостей

В данном диалоге можно задать перечень запрещенных зависимостей, которые можно добавлять с помощью кнопки **Add** (Добавить), удалять с помощью кнопки **Remove** (Удалить), а также перемещать вверх и вниз в таблице с помощью кнопок **Move Up/Down** (Переместить вверх/вниз).

После добавления правила проверки в ячейках таблицы **Deny usages of** (Запретить использование) **in** (в) появятся выпадающие списки областей классов (**Scope**).

Продемонстрируем это на примере. Допустим, мы хотим, чтобы классы одного из модулей нашего проекта не использовали классы из пакета `java.util`. Настроим соответствующие наборы классов.

Области классов настраиваются с помощью диалога **Edit Scopes** (Редактирование областей) (рис. 6.44).

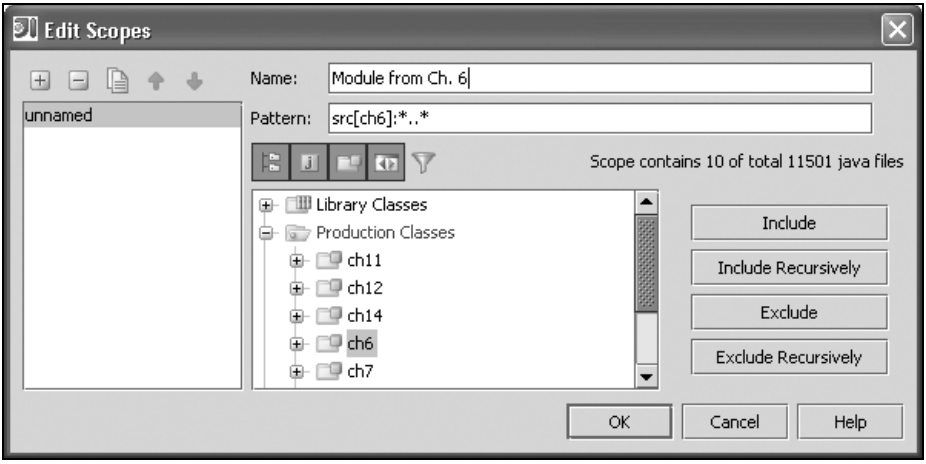


Рис. 6.44. Редактирование областей классов

В данном диалоге содержится перечень областей, которые можно добавлять (кнопка **Add**), удалять (кнопка **Remove**), копировать (кнопка **Copy**), а также менять порядок (кнопки **Move Up/Down**).

Область классов имеет следующие параметры:

- **Name** (Имя) — имя набора классов;
- **Pattern** (Образец выборки классов) — строка со сложным синтаксисом, которую лучше не редактировать вручную.

Образец выборки классов можно создавать с помощью перечня классов, представленных в диалоге, которые можно добавлять в область с помощью кнопок **Include** (Включить) и **Include Recursively** (Включить рекурсивно), а также исключать из области с помощью кнопок **Exclude** (Исключить) и **Exclude Recursively** (Исключить рекурсивно).

Мы создали два образца:

- Классы модуля `ch6` - `src[ch6]:*..*`.
- Классы из пакета `java.util`, а также из его вложенных пакетов — `lib:java.util..*`.

Таким образом, мы можем задать необходимое нам правило (рис. 6.45).

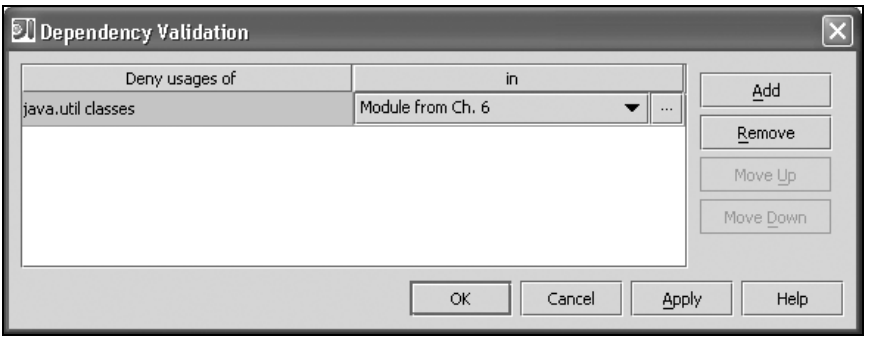


Рис. 6.45. Правило задано

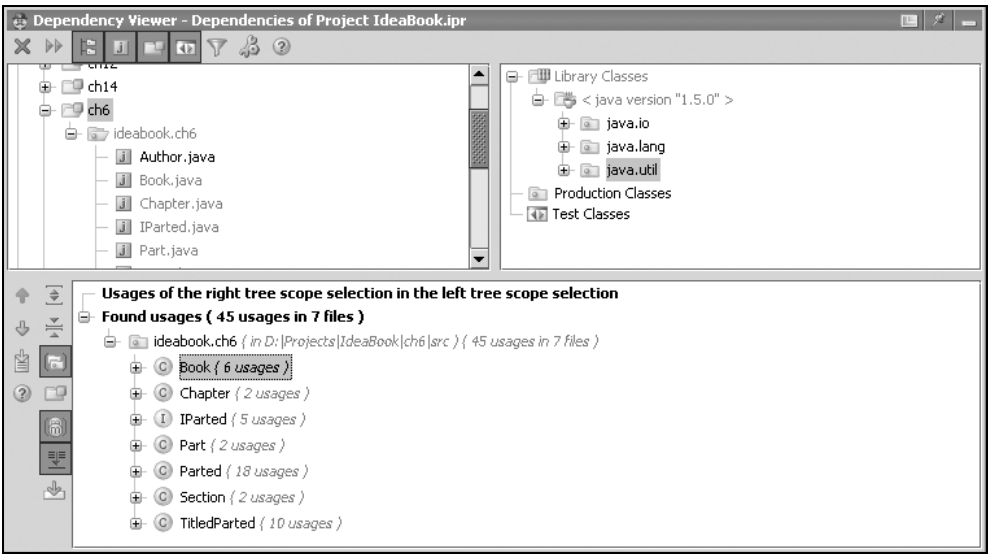


Рис. 6.46. Найдены запрещенные зависимости

После того как правило было задано, команда анализа зависимостей обнаружила зависимости между классами модуля `ch6` и пакета `java.util`, которые она выделила красным цветом (рис. 6.46), к сожалению, на черно-белой иллюстрации это не видно.

6.4.3. Анализ дубликатов

Анализ дубликатов или повторов кода позволяет выявить код в конкретном классе, пакете, модуле или во всем проекте, который с той или иной степенью "похожести" встречается в нескольких классах (*см. раздел 3.2.7.4*).

В больших проектах часто бывает, что одни и те же алгоритмы пишутся несколькими людьми независимо друг от друга, либо код копируется между классами. Все это нарушает принципы ООП.

Чтобы избежать возможных проблем, к которым приводит дублирование кода, можно периодически проводить анализ дубликатов кода с помощью соответствующей команды.

Перед выполнением команда анализа зависимостей открывает диалог с настройками поиска, представленный на рис. 6.47.

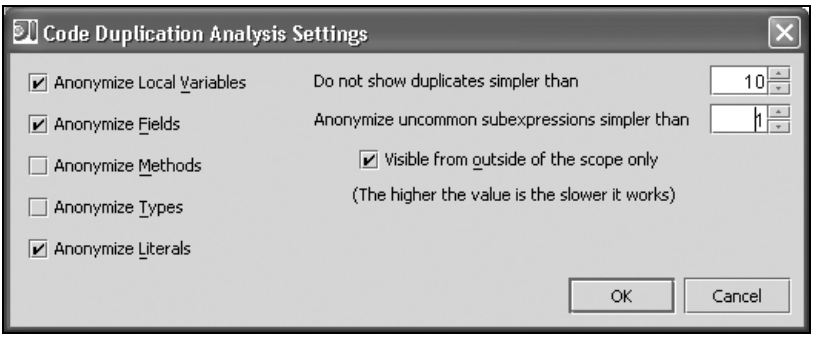


Рис. 6.47. Настройки поиска зависимостей

В данном диалоге представлены следующие опции и параметры:

- **Anonymize Local Variables** (Анонимизировать локальные переменные) — не различать имена локальных переменных, т. е. обращать внимание не на имена, а на конкретное использование;
- **Anonymize Fields** (Анонимизировать поля касса) — не различать имена полей класса;
- **Anonymize Methods** (Анонимизировать методы) — не различать имена методов. Данная опция по умолчанию выключена;
- **Anonymize Types** (Анонимизировать типы) — не различать имена типов. Данная опция по умолчанию выключена;
- **Anonymize Literals** (Анонимизировать литералы) — анонимизировать строки;
- **Do not show duplicates simpler than** (Не показывать дубликаты проще, чем) — некоторая эвристическая мера похожести фрагментов кода;

- **Anonymize uncommon subexpressions simpler than** (Не различать строковые выражения проще, чем). Если данный параметр отличен от нуля, имеется возможность задать дополнительную опцию **Visible from outside of the scope only** (Только видимые извне текущей области видимости).

Чем больше эвристические значения, тем медленнее будет работать анализ дубликатов.

Результат анализа выводится в инструментальном окне **Find** (Найти) (*см. раздел 3.4.8.4*). Общий вид результатов поиска представлен на рис. 6.48.

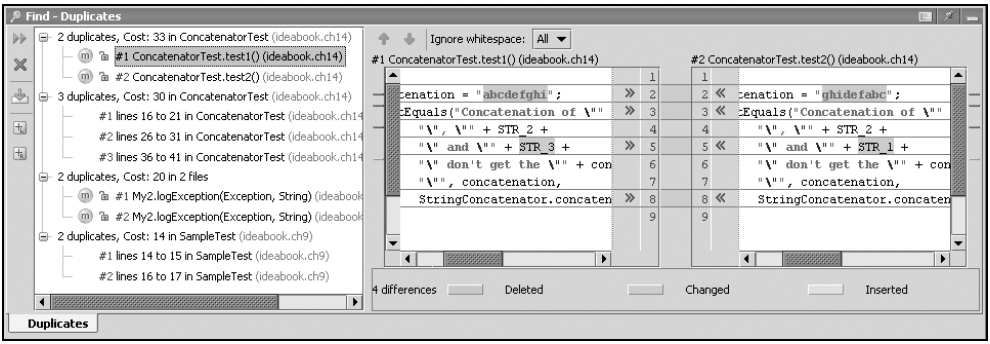


Рис. 6.48. Результаты поиска дубликатов

В левой части инструментального окна выводится общий список найденных дубликатов. В списке указывается количество найденных дубликатов, эвристическая оценка "похожести", а также название класса, в котором они расположены.

Правая часть окна состоит из панели сравнения (*см. раздел 12.1*), в левой и правой части которой представлены похожие фрагменты.

На рис. 6.48 указанные фрагменты отличаются лишь конкретными значениями строк, а также параметрами, передаваемыми функции. Очевидно, что данные фрагменты можно заменить вызовом метода, который бы получал необходимые параметры.

На панели инструментов окна имеются две дополнительные кнопки:

- **Send to Left** (Отправить налево) (<Ctrl>+<1>) — поместить указанный фрагмент на левую панель для сравнения;
- **Send to Right** (Отправить направо) (<Ctrl>+<2>) — поместить указанный фрагмент на правую панель для сравнения.

Данные кнопки удобно использовать, если некоторый фрагмент найден более чем в двух местах кода.

6.5. Структурный поиск

Функция структурного поиска позволяет выполнять поиск фрагментов Java-кода, основанный не только на простом совпадении фрагментов текста в некоторой подстроке (или регулярном выражении), но также и на основе образца структуры кода.

Допустим, нам нужно найти все классы, которые имплементируют хотя бы один из заданных интерфейсов. Мы могли бы последовательно произвести поиск имплементаций для каждого из интерфейсов, а затем сравнить результаты, но это может занять довольно много времени.

С помощью функции структурного поиска такой поиск можно осуществить довольно просто и быстро.

Структурный поиск запускается с помощью команды **Search Structurally** (Структурный поиск) (<Ctrl>+<Shift>+<S>). Данная команда открывает диалог структурного поиска (рис. 6.49).

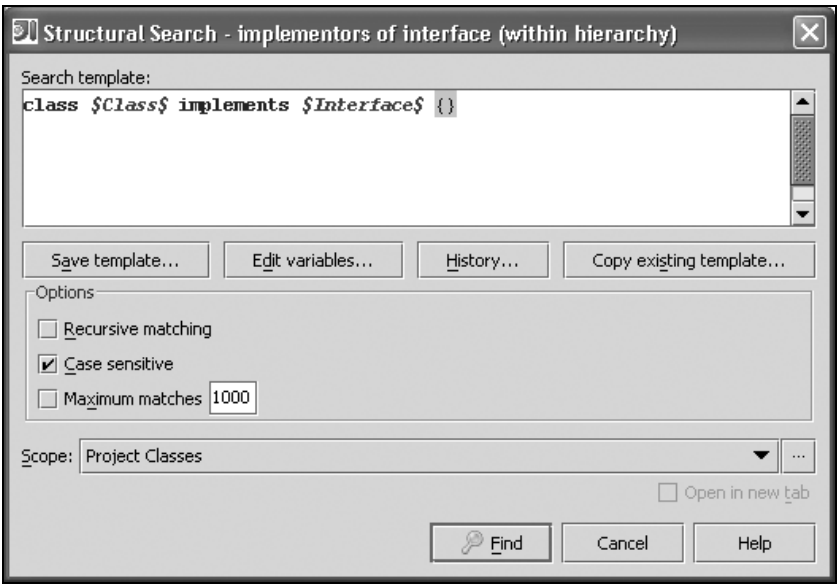


Рис. 6.49. Диалог структурного поиска

В данном диалоге можно задать следующие параметры и опции:

- **Search template** (Шаблон для поиска);
- **Options** (Опции):
  - **Recursive matching** — искать рекурсивно, в том числе и в найденных результатах;

- **Case sensitive** — распознавать регистр символов;
- **Maximum matches** — максимальное количество найденных результатов;

❐ **Scope** (Область поиска) — в качестве области поиска можно выбрать одну из указанных областей:

- **Project Classes** (Классы проекта);
- **Project and Libraries** (Проект и библиотеки) — все классы из проекта и из библиотек;
- **Project Production Classes** (Классы проекта без учета тестов);
- **Project Test Classes** (Классы тестов проекта);
- **Current File** (Текущий файл);
- **Class Hierarchy** (Иерархия класса).

Если ни одна из указанных областей не подходит, можно создать собственную, нажав на кнопку со знаком "...". Данная кнопка открывает диалог настройки областей класса (см. рис. 6.44).

С шаблонами поиска можно производить следующие действия:

❐ **Save template** (Сохранить шаблон) — сохранить шаблон в список доступных шаблонов;

❐ **Edit variables** (Редактировать переменные шаблона);

❐ **History** (История использования шаблонов);

❐ **Copy existing template** (Копировать существующий шаблон).

Все эти действия можно выполнить с помощью соответствующих кнопок.

Шаблон структурного поиска чем-то напоминает "живые шаблоны" (*см. раздел 4.1.13*), если не функциональностью, то по крайней мере формой представления — здесь также имеются переменные, которые несут некоторую функциональную нагрузку.

6.5.1. Редактирование переменных

Редактирование переменных производится с помощью диалога, представленного на рис. 6.50.

Данный диалог содержит список доступных переменных шаблона (**Variables**) — буквенных последовательностей, заключенных между символами '$'.

Для каждой переменной можно задать следующие опции и параметры:

❐ **Text constraints** (Текстовые ограничения):

- **Text / regular expression** (Текст / регулярное выражение) — текст или регулярное выражение, которому должна соответствовать указанная переменная;

- **Invert condition** (Инвертировать условие);
- **Apply constraint within type hierarchy** (Применить ограничение внутри иерархии типов) — данная опция играет роль, если переменная задает имя некоторого типа. Если опция включена, поиск будет осуществлен не только в пределах конкретного типа, но также и у его базовых типов (в иерархии типов);
- **Whole words only** (Слова только целиком) — находить слова, соответствующие указанному тексту целиком (исключая фрагменты);

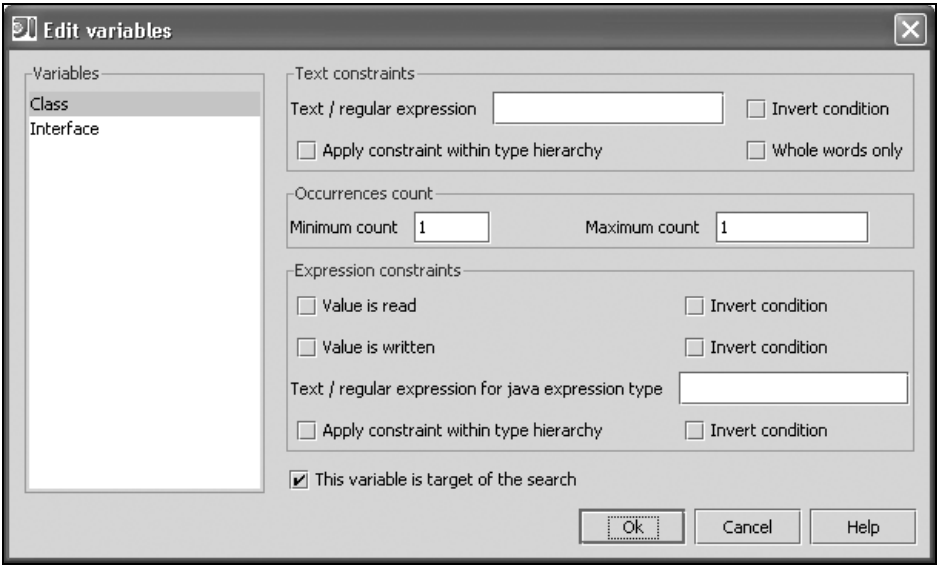


Рис. 6.50. Диалог редактирования переменных шаблона структурного поиска

- **Occurrences count** (Количество вхождений) — количество последовательных элементов (параметров, определений или операторов), которое может включать переменная:
  - **Minimum count** (Минимальное количество);
  - **Maximum count** (Максимальное количество);
- **Expression constraints** (Ограничения выражения) — ограничения для переменных, задающих некоторое выражение:
  - **Value is read** (Значение читается);
  - **Invert condition** (Инвертировать условие) — инвертировать предыдущее условие;
  - **Value is written** (Значение записывается);

- **Invert condition** (Инвертировать условие) — инвертировать предыдущее условие;
- **Text / regular expression for java expression type** (Текст / регулярное выражение для Java-типа выражения);
- **Invert condition** (Инвертировать условие) — инвертировать предыдущее условие;
- **Apply constraint within type hierarchy** (Применить ограничение внутри иерархии типов) — данная опция играет роль, если переменная задает имя некоторого типа. Если опция включена, поиск будет осуществлен не только в пределах конкретного типа, но также и у его базовых типов (в иерархии типов);

❐ **This variable is target of the search** (Эта переменная является целью поиска).

6.5.2. Доступные шаблоны

IntelliJ IDEA имеет предопределенный набор шаблонов, которые можно использовать для поиска, а также в качестве образцов для создания собственных шаблонов. Предопределенные шаблоны представлены в диалоге на рис. 6.51.

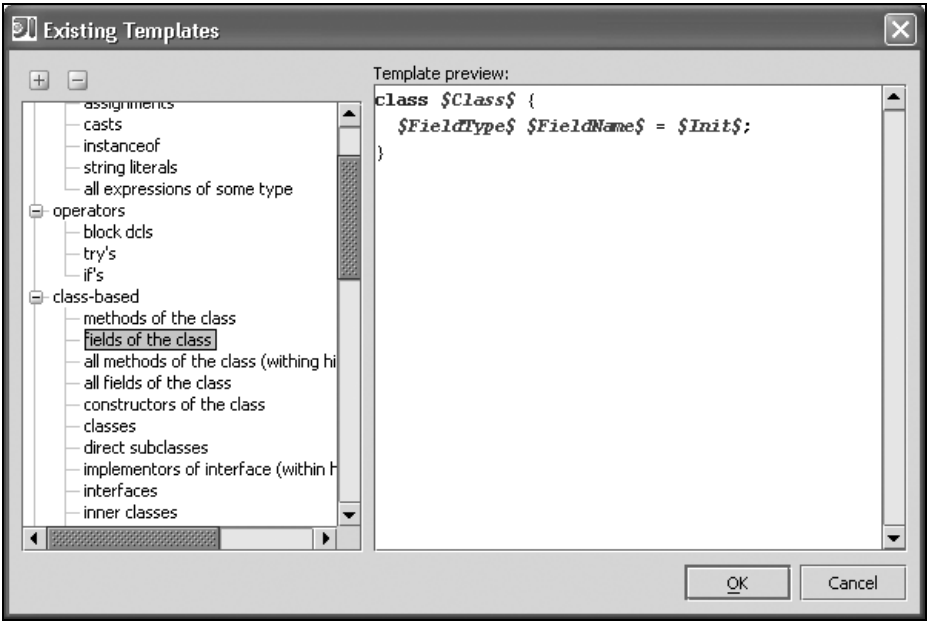


Рис. 6.51. Доступные шаблоны

В левой части диалога представлен список шаблонов, в правой части — панель просмотра шаблона.

Доступные шаблоны сгруппированы в следующие группы:

- **expressions** — выражения;
- **operators** — операторы;
- **class-based** — основанные на классах;
- **generics** — генерические;
- **comments, javadoc and metadata** — комментарии JavaDoc и метаданные;
- **j2ee**;
- **misc** — прочие;
- **interesting** — интересные;
- **user defined** — определенные пользователем.

6.5.3. Результаты поиска

Результаты структурного поиска отображаются в инструментальном окне **Find** (Поиск) (*см. раздел 3.4.8.4*).

6.5.4. Структурная замена

Структурная замена производится с помощью диалога, представленного на рис. 6.52.

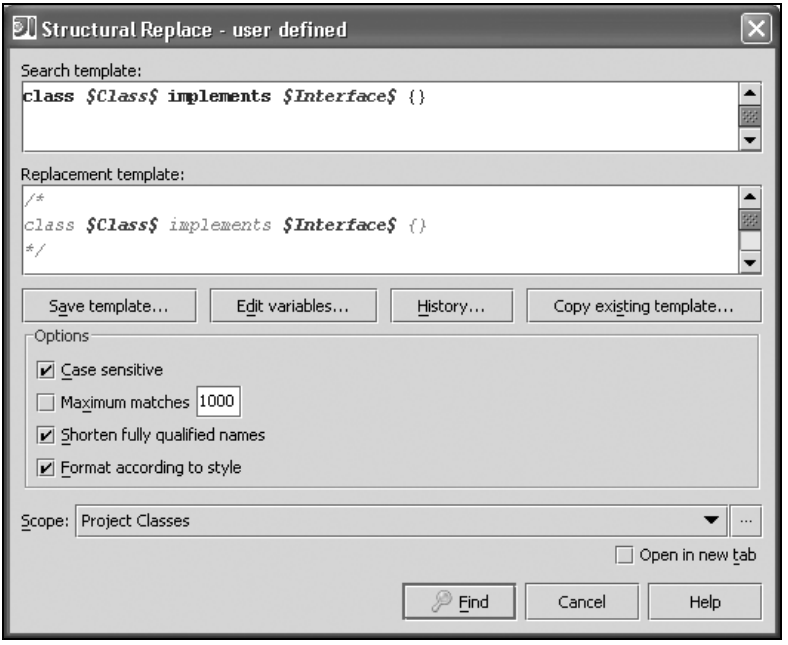
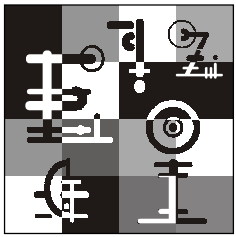


Рис. 6.52. Диалог структурной замены

Данный диалог немного отличается от диалога структурного поиска и имеет следующие дополнительные параметры и опции:

- **Replacement templat**e (Шаблон для замены) — текстовое выражение, которое будет использовано для замены найденных фрагментов кода. Данный шаблон может использовать переменные, но только те, что заданы в шаблоне поиска;
- **Shorten fully qualified names** (Кратчайшие полные имена) — данная опция позволяет автоматически сокращать полностью заданные мена классов при замене;
- **Format according to style** (Форматировать согласно стилю) — форматировать результат замены согласно установленному стилю (*см. раздел 4.1.7*).



Глава 7

Создание интерфейса пользователя

Многие известные интегрированные среды разработки программного обеспечения (IDE — Integrated Development Environment) на Java, например такие, как Borland JBuilder или Visual J++, ориентированы на создание визуального интерфейса приложения. В IntelliJ IDEA изначально основной упор был сделан не на визуальные средства, а на удобство создания кода. Потому что при создании интерфейса пользователя, несмотря на все визуальные редакторы, сколь бы хорошими и удобными они не были, все равно приходится дописывать код вручную.

Начиная с версии 4, IntelliJ IDEA получила свой визуальный редактор.

В основе концепции создания пользовательского интерфейса в IntelliJ IDEA лежит компонентная модель библиотеки Swing (javax.swing). IntelliJ IDEA использует базовый набор компонент Swing.

Для задания взаимного расположения компонент в Swing используются специальные Layout-объекты из Java AWT (Abstract Window Toolkit).

В отличие от многих других IDE, например Borland JBuilder, в IntelliJ IDEA использована довольно интересная идеология создания пользовательского интерфейса. В IntelliJ IDEA нет поддержки всех возможных Layout-объектов. Вместо этого, IntelliJ IDEA создает собственный Layout-объект, который встраивается в *форму* пользовательского интерфейса. Затем форма преобразуется в Java-код, инициализирующие компоненты формы.

Расположение компонент основано на табличной верстке с дополнением из *распорок* — специальных невидимых компонент.

7.1. Формы

Форма — это единица пользовательского интерфейса в IntelliJ IDEA.

Форма — это файл в формате XML, который задает взаимное расположение компонент. Форма также является надстройкой над обычным Java-классом,

устанавливает ассоциации между полями этого класса и компонентами формы. Все компоненты формы объединены в иерархические структуры, в основе которых может находиться один из `Swing`-контейнеров, чаще всего это `JPanel`.

7.1.1. Создание формы

Разобраться в сущности форм проще всего на примере. Для начала нужно создать файл с пустой формой. Это можно сделать, вызвав команду **Edit | New**

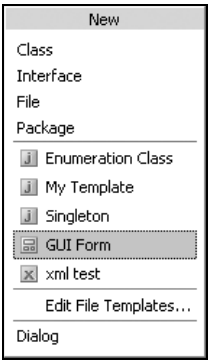


Рис. 7.1. Выбор пункта **GUI Form** в меню **Edit | New**

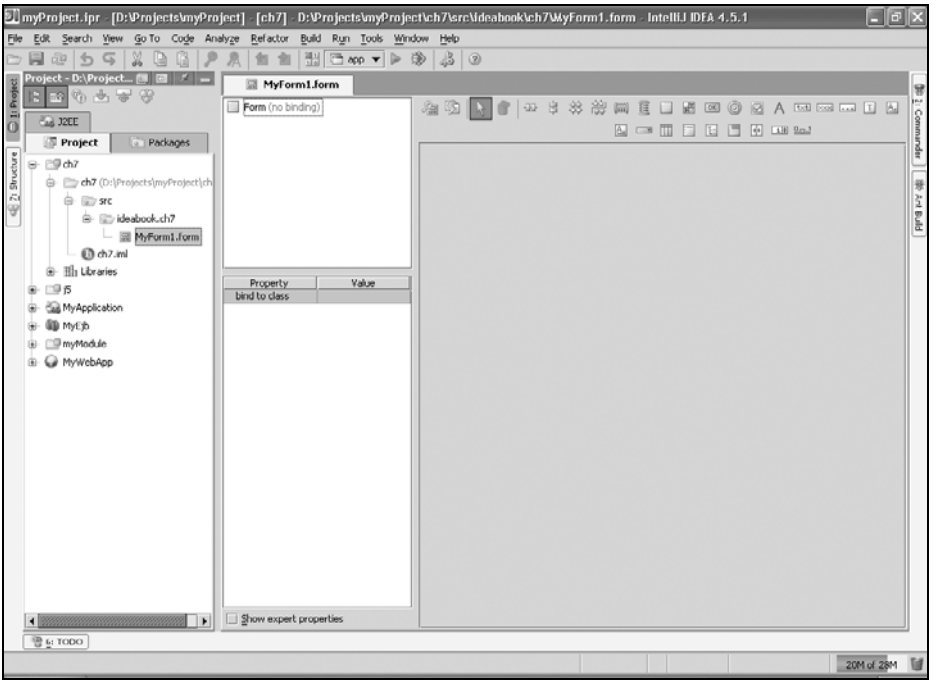


Рис. 7.2. Форма, открытая в визуальном редакторе IntelliJ IDEA

(Редактирование | Создать) (*см. раздел 3.2.2.10*) внутри одного из путей с исходными текстами (*см. раздел 5.4.1*).

В качестве объекта для создания нужно выбрать пункт **GUI Form** (Форма пользовательского интерфейса) (рис. 7.1).

Затем нужно ввести имя файла формы, после чего IntelliJ IDEA откроет созданную форму в визуальном редакторе (рис. 7.2).

Визуальный редактор размещается на одной из закладок с редакторами IntelliJ IDEA (*см. раздел 3.6*), но вместо текста он имеет три части:

- часть со структурой формы, размещенная в левом верхнем углу;
- часть со свойствами элемента формы, размещенная в левом нижнем углу;
- поле размещения компонент формы, занимающее всю оставшуюся часть закладки редактора. В ее верхней части имеется панель инструментов.

7.1.1.1. Панель инструментов

На панели инструментов формы (рис. 7.3) содержатся как команды управления разметкой, так и команды создания тех или иных компонент.

Рис. 7.3. Панель инструментов формы

Рассмотрим все команды (слева направо):

- Команды просмотра и редактирования связей:
  - **Preview** (Предварительный просмотр) — команда позволяет выполнить предварительный просмотр формы;
  - **Data Binding Wizard** (Мастер связей) — команда запускает мастер связей формы (*см. раздел 7.4.2*);
- Команды разметки:
  - **Lay Out Horizontally** (Разместить горизонтально) — команда выполняет горизонтальное размещение выделенных компонент;
  - **Lay Out Vertically** (Разместить вертикально) — команда выполняет вертикальное размещение выделенных компонент;
  - **Lay Out In a Grid** (Разместить в сетке) — команда размещает выделенные компоненты в сетке;
  - **Break Layout** (Убрать размещение) — команда снимает правила размещения компонент;

- Команды создания компонент:
  - Специальные компоненты:
    - ◊ **Horizontal Spacer** — горизонтальная распорка (компонента с пиктограммой в виде горизонтальной пружинки);
    - ◊ **Vertical Spacer** — вертикальная распорка (компонента с пиктограммой в виде вертикальной пружинки);
  - Стандартные компоненты `Swing`:
    - ◊ **JPanel (javax.swing)** — панель;
    - ◊ **JScrollPane (javax.swing)** — панель с полосами прокрутки;
    - ◊ **JButton (javax.swing)** — кнопка;
    - ◊ **JRadioButton (javax.swing)** — переключатель ("радиокнопка");
    - ◊ **JCheckBox (javax.swing)** — опция-переключатель;
    - ◊ **JLabel (javax.swing)** — текстовая метка;
    - ◊ **JTextField (javax.swing)** — поле ввода текста;
    - ◊ **JPasswordField (javax.swing)** — поле ввода пароля;
    - ◊ **JFormattedTextField (javax.swing)** — поле ввода форматированного текста;
    - ◊ **JTextArea (javax.swing)** — многострочное поле ввода текста;
    - ◊ **JTextPane (javax.swing)** — текстовое поле с разметкой;
    - ◊ **JEditorPane (javax.swing)** — расширенное текстовое поле с разметкой;
    - ◊ **JComboBox (javax.swing)** — выпадающий список;
    - ◊ **JTable (javax.swing)** — таблица;
    - ◊ **JList (javax.swing)** — список;
    - ◊ **JTree (javax.swing)** — дерево;
    - ◊ **JTabbedPane (javax.swing)** — панель с закладками;
    - ◊ **JSplitPane (javax.swing)** — панель с разделителем;
    - ◊ **JSpinner (javax.swing)** — поле ввода с кнопками инкремента/декремента;
    - ◊ **JSlider (javax.swing)** — поле выбора значения из заданного диапазона.

7.1.1.2. Размещение компонент

После того как форма создана, в ней можно разместить компоненты.

Создадим простой диалог аутентификации пользователя с логином и паролем (рис. 7.4).

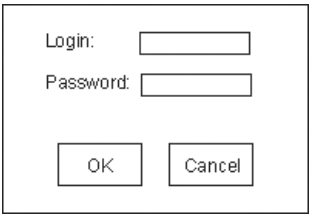


Рис. 7.4. Набросок диалога

Для этого нам понадобятся следующие компоненты:

- `JLabel` — 2 шт.;
- `JTextField` — 1 шт.;
- `JpasswordField` — 1 шт.;
- `JButton` — 2 шт.

Чтобы разместить компоненты в форме, нужно щелкнуть мышью кнопку панели инструментов, соответствующую компоненте. Затем нужно щелкнуть левой кнопкой в некотором месте в форме. Компонента разместится в этом месте (рис. 7.5).

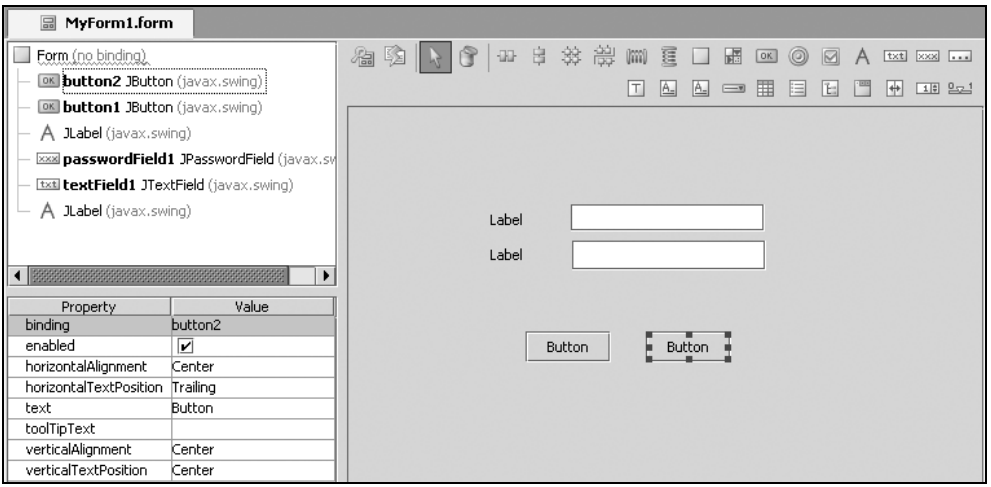


Рис. 7.5. Размещение компонент в форме

Примечание

Если размещать компоненты в форме, держа нажатой клавишу <Ctrl>, можно разместить несколько однотипных компонент.

С редактором формы произошли некоторые изменения.

Во-первых, в поле компонент появились необходимые нам компоненты. Во-вторых, в поле структуры формы появились соответствующие записи о размещенных компонентах.

Теперь нам нужно задать текстовое содержание для двух меток (**Login**, **Password**), а также для кнопок (**Log In**, **Cancel**).

Есть два способа задать текстовое значение для компоненты:

- ❒ непосредственно;
- ❒ через связь с файлом ресурсов (ResourceBundle) (*см. раздел 7.4.4*).

Для начала будем использовать самый простой способ задания значений непосредственно.

Если дважды щелкнуть левой кнопкой мыши на компоненте, отобразится поле ввода, в котором можно задать текстовое значение компоненты (рис. 7.6).

Аналогично устанавливаем значения для всех компонент (рис. 7.7).

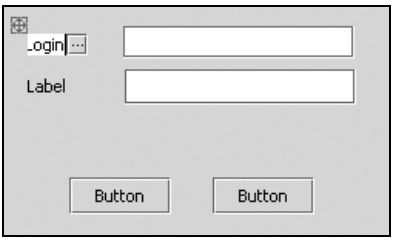


Рис. 7.6. Установка текстового значения компоненты

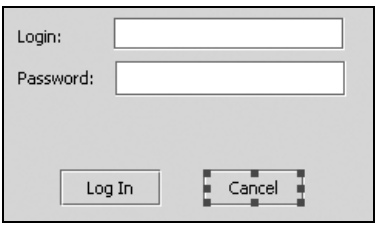


Рис. 7.7. Все текстовые значения установлены

Идем дальше. Обратите внимание на волнистое подчеркивание в поле структуры формы (в левой верхней части рис. 7.5) — это индикатор того, что чего-то не хватает. Если подвести курсор мыши к подчеркнутому элементу, во всплывающем окошке появится сообщение об ошибке (рис. 7.8).

Сообщение гласит: **Form cannot be compiled because it contains more than one component at the top level** — форма не может быть скомпилирована, поскольку содержит более одной компоненты верхнего уровня.

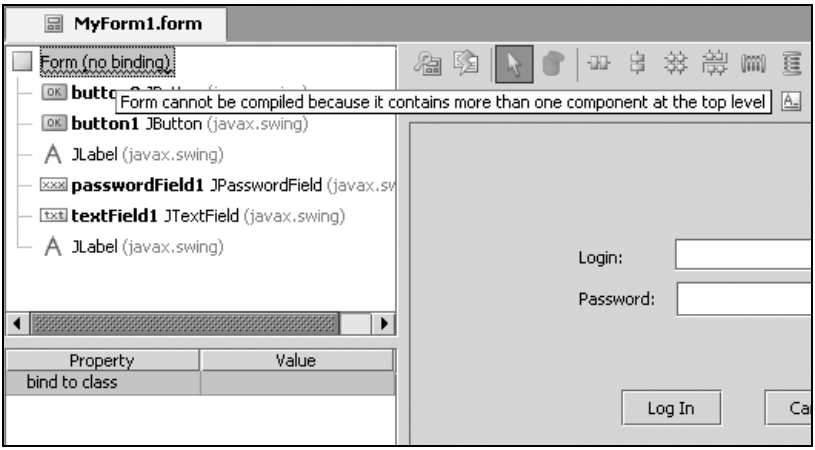


Рис. 7.8. Сообщение об ошибке

Примечание

Это сообщение отражает одну из особенностей создания визуальных форм в IntelliJ IDEA: форма имеет строго иерархическое представление с одной корневой компонентой.

Чтобы исправить ситуацию, нужно установить размещение компонент по контейнерам с помощью горизонтальной, вертикальной или сеточной разметки.

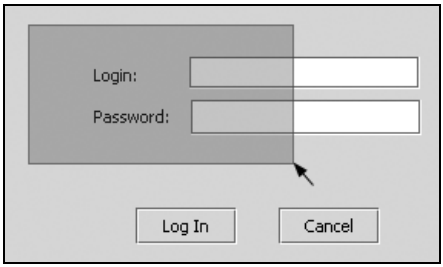


Рис. 7.9. Выделение компонент формы

Сделаем следующее:

1. Выделим обе метки и поля для ввода. Выделенные компоненты будут иметь обозначенные метками (синего цвета) границы (рис. 7.10).

Примечание

Выделение можно выполнить как с помощью последовательного нажатия на компоненты левой кнопкой мыши совместно с нажатой клавишей <Ctrl>, так

и щелкнув в некоторую пустую область формы, нажав левую кнопку мыши, а затем перемещая мышь (см. рис. 7.9). В визуальном редакторе также работает команда **Select Word At Caret** (<Ctrl>+<W>), которая позволяет выделять компоненты относительно их иерархического размещения в контейнерах.

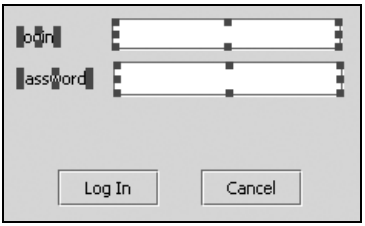


Рис. 7.10. Выделенные компоненты формы

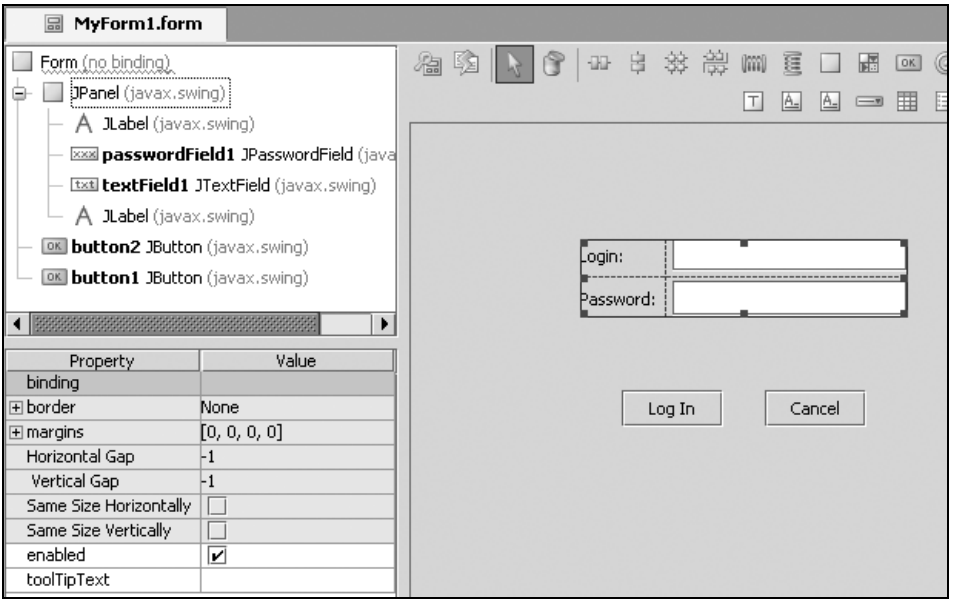


Рис. 7.11. Компоненты размещены внутри сеточного контейнера

2. Нажмем кнопку **Lay Out In a Grid** (Разместить в сетке), после чего выделенные компоненты окажутся размещенными в специальном сеточном контейнере (рис. 7.11). Обратите внимание на поле структуры формы. В нем появилась дополнительная компонента `JPanel`, которая стала объединять четыре компоненты, размещенные в сетке.
3. Прежде чем заняться размещением компонент с кнопками, добавим в форму еще три дополнительных компоненты. Их присутствие несет особую функциональность, которая позволит нам существенно улучшить

визуальное представление нашего диалога. Разместим в нашей форме две горизонтальные распорки и одну вертикальную так, как это показано на рис. 7.12. Мы добавили распорки, чтобы компоненты формы вплотную не прилегали друг к другу. Распорки — это функциональные элементы формы, не имеющие визуального представления.

4. Выделим компоненты кнопок и двух горизонтальных распорок и выполним для них команду **Lay Out Horizontally** (Разместить горизонтально) (рис. 7.13).
5. Объединим две группы компонент и одну вертикальную распорку между ними с помощью команды **Lay Out Vertically** (Разместить вертикально) (рис. 7.14).

Теперь ошибка из поля структуры формы исчезла, а все компоненты формы оказались объединены в иерархическую структуру из `JPanel`-контейнеров.

Можно запустить предварительный просмотр командой **Preview** (Предварительный просмотр) (рис. 7.15).

Размещение компонент в диалоге адекватно тому, чего мы и хотели добиться. Распорки тоже работают, достаточно попробовать изменить размер окна (рис. 7.16).

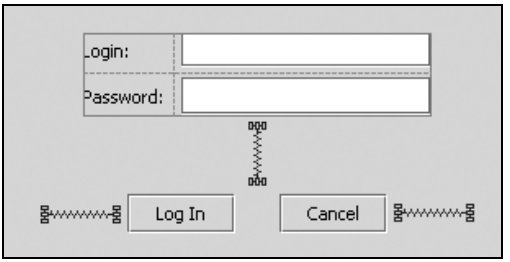


Рис. 7.12. Размещение в форме горизонтальных и вертикальных распорок

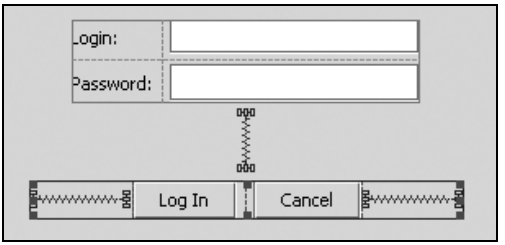


Рис. 7.13. Горизонтальное размещение для компонент кнопок и распорок

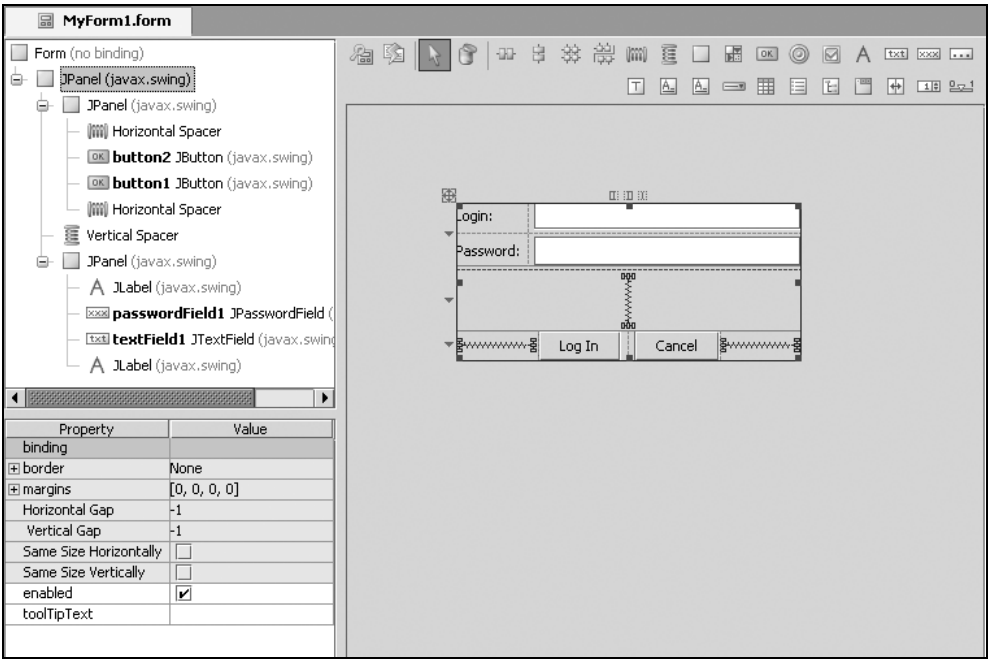


Рис. 7.14. Все компоненты формы объединены иерархически с помощью размещения в контейнерах

Рис. 7.15. Предварительный просмотр нашего диалога

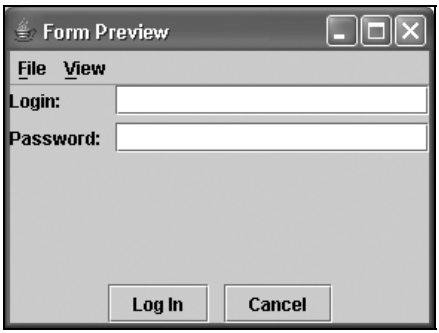


Рис. 7.16. Изменение размера диалога: распорки работают

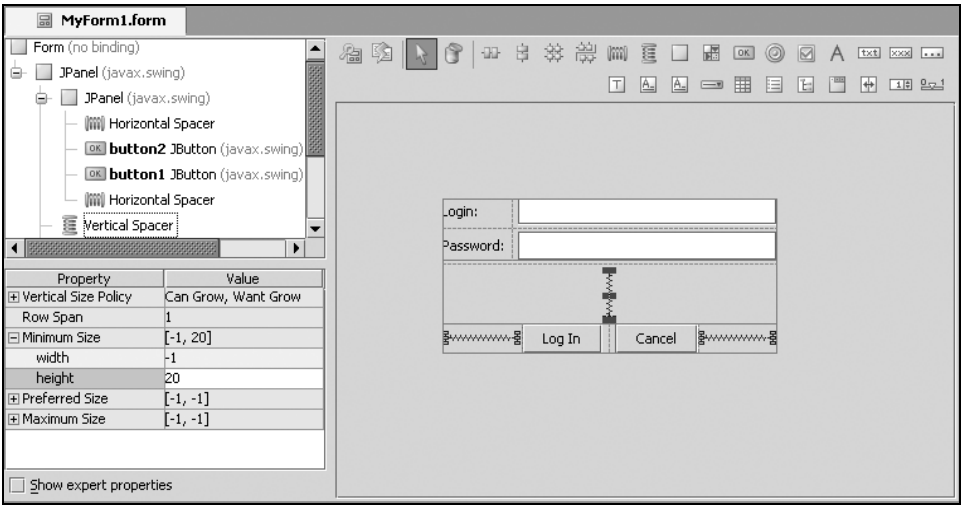


Рис. 7.17. Установка минимальной высоты вертикальной распорки

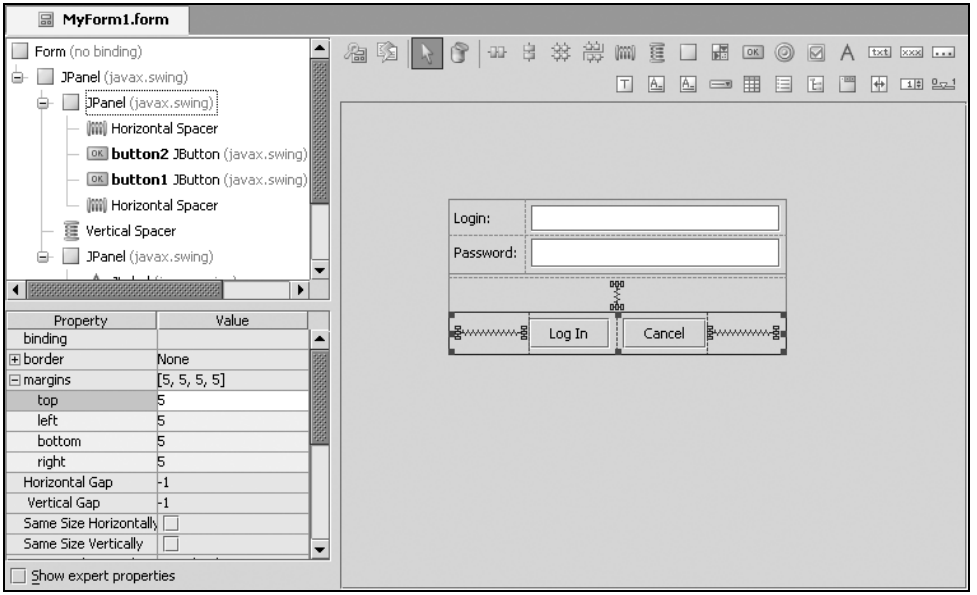


Рис. 7.18. Установка границ для сетки с метками и полями ввода и для сетки с кнопками

Нужно сделать еще небольшое изменение, чтобы с размещением совсем все было хорошо:

1. Выделить вертикальную распорку и в поле свойств компоненты установить **Minimum Size: width** (Минимальный размер: ширина) = 20, чтобы кнопки немного отстояли от полей ввода (рис. 7.17). Данная распорка не

только позволяет визуально разделить блок ввода и блок кнопок, но также "приклеит" кнопки к низу формы. Если требуется просто установить фиксированный отступ одной компоненты от другой, можно также использовать границы (**margins**) (*см. следующий пункт*).

2. Выделить сетку, в которой размещены метки и поля ввода, и установить для нее в свойствах компоненты границы (**margins**), каждая из которых равна пяти (**top**, **left**, **bottom**, **right**). Это позволит немного раздвинуть компоненты и сделает форму более симпатичной.
3. Аналогично зададим границы (**margins**) для сетки с кнопками и горизонтальными распорками (рис. 7.18).

С помощью команды **Preview** (Предварительный просмотр) можем убедиться, что диалог стал выглядеть лучше.

7.1.1.3. Привязка формы к классу

Прежде чем использовать нашу форму, нужно сделать последнее — связать ее с некоторым классом, который будет выполнять всю дополнительную функциональность по работе формы.

Так как у нас нет готового класса, с которым мы могли бы связать форму, его нужно создать с помощью команды **Edit | New** (Редактирование | Создать). Назовем его `My1` (рис. 7.19).

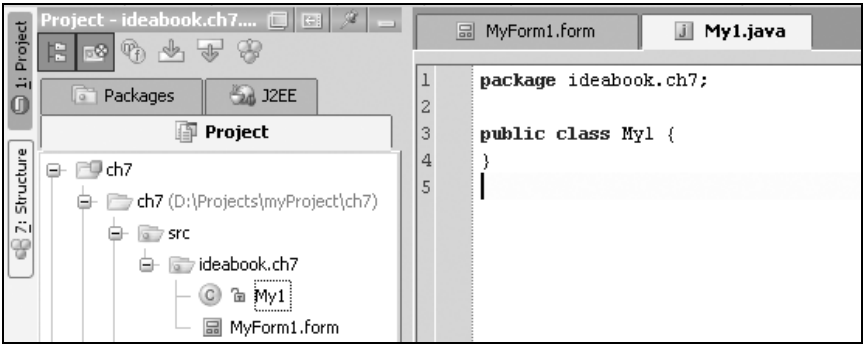


Рис. 7.19. Создание класса для привязки к нему формы

Теперь переключимся на закладку с формой и переместим подсветку в поле структуры формы к самому верхнему узлу **Form**. Справа от него в скобках написано **no binding** — это означает, что наша форма не связана ни с одним классом. В поле свойств щелкнем левой кнопкой мыши на команде **bind to class** (рис. 7.20) и в диалоге выбора класса найдем созданный нами класс (рис. 7.21).

Класс `My1` будет связан с формой (рис. 7.22).

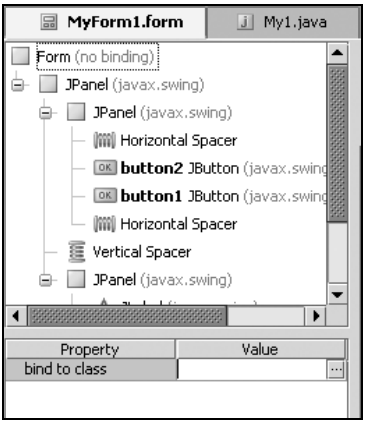


Рис. 7.20. Выбор класса для связи с формой: шаг 1

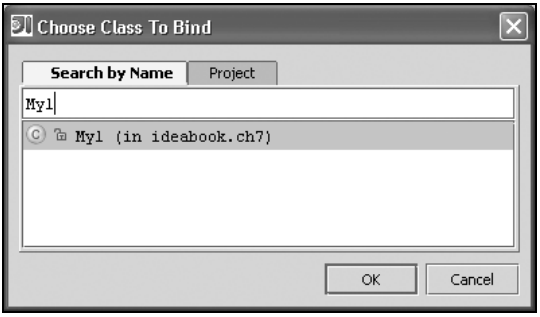


Рис. 7.21. Выбор класса для связи с формой: шаг 2

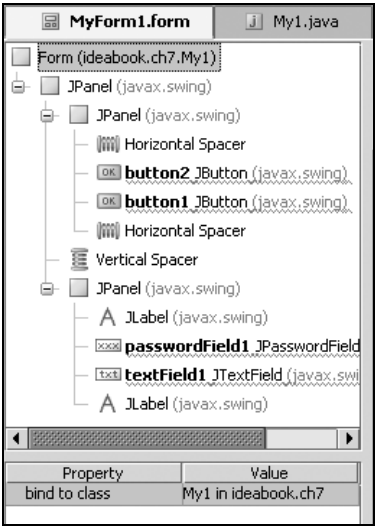


Рис. 7.22. Класс `My1` связан с формой

Примечание

Форму можно связывать также и со внутренним классом какого-либо класса. Сделать это можно аналогично, с помощью диалога, представленного на рис. 7.21.

В поле структуры формы снова появилось красное волнистое подчеркивание, что свидетельствует о том, что для формы требуется еще что-то сделать. Если подвести курсор мыши к выделенной компоненте, можно будет прочитать сообщение об ошибке: **Field 'button2' doesn't exists in class "ideabook.ch7.My1"** — поле `button2` не существует в классе `My1`, что вполне справедливо.

Названия наших компонент, которые мы размещали в форме, были заданы автоматически. Прежде чем связывать поля класса с компонентами в форме, дадим им понятные имена (рис. 7.23).

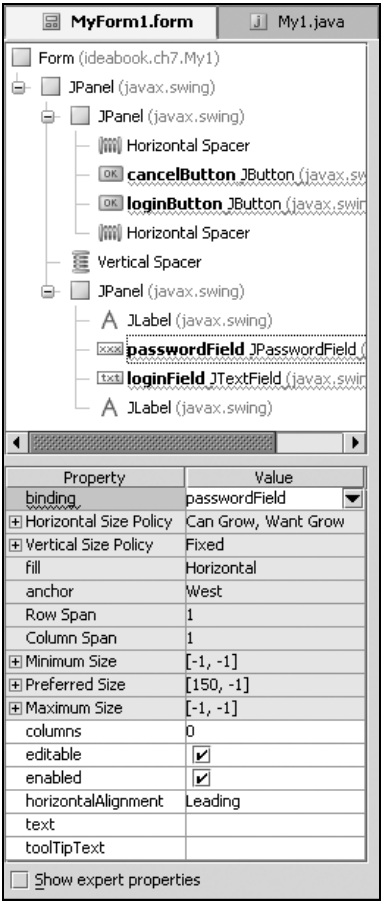


Рис. 7.23. Компоненты формы получили понятные названия

Теперь у нас есть два пути:

1. Создать необходимые поля класса, соответствующие компонентам в форме вручную, добавив их в определение класса `My1`.
2. Воспользоваться "подсказками" IntelliJ IDEA, подведя курсор к названию компоненты в поле структуры формы и нажав на значок лампочки, который появится рядом с ним (рис. 7.24) (подробнее о "подсказках" IntelliJ IDEA можно прочитать в *разделе 6.1.3*).

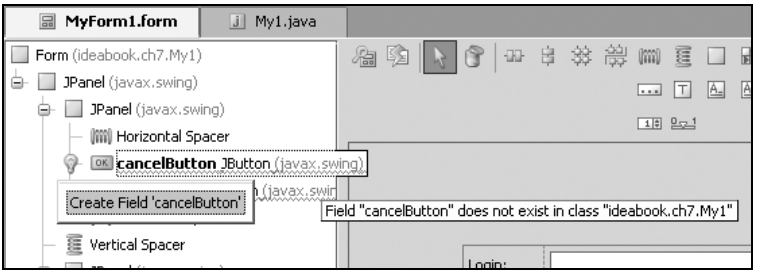


Рис. 7.24. "Подсказка" IntelliJ IDEA о создании поля класса для компоненты

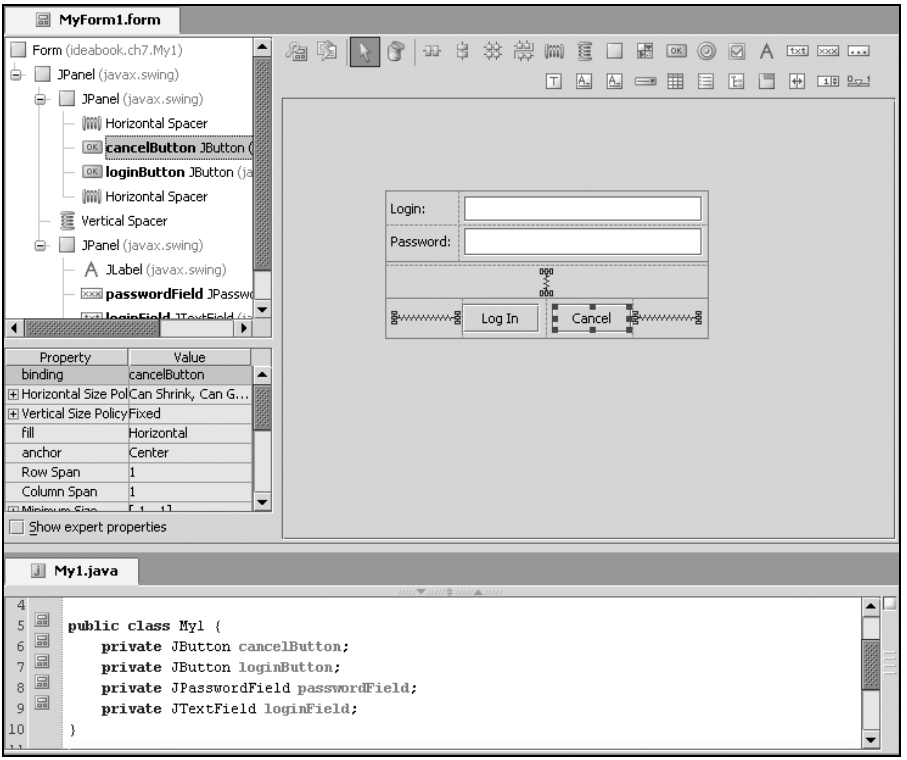


Рис. 7.25. Поля класса для компонент формы созданы

Для нового класса мы предлагаем воспользоваться вторым вариантом, как более быстрым и надежным. Выполнив действия "подсказок" для каждой компоненты, IntelliJ IDEA создаст четыре поля класса `My1` (рис. 7.25).

Примечание

Для создания рис. 7.25 мы воспользовались возможностью разделения закладок с редакторами (*см. раздел 3.2.1.10*).

Осталось выполнить последнее действие, и нашу форму можно будет использовать. Оно несколько неочевидное, за что многие поклонники других визуальных редакторов (например, Borland JBuilder) часто ругают IntelliJ IDEA.

Обратите внимание на значки, появившиеся на полях редактора с классом `My1`. Они свидетельствуют о том, что данный класс, а также помеченные поля класса связаны с формой. Если щелкнуть на этих значках левой кнопкой мыши, IntelliJ IDEA автоматически откроет для нас редактор с формой.

Нам нужно создать привязку самой верхней панели в иерархической структуре формы к полю класса `My1`. Назовем ее `mainPanel` (рис. 7.26).

Это было нужно сделать для того, чтобы иметь какую-то "зацепку" (в виде поля класса) для созданной нами формы.

Теперь можем добавить в класс `My1` тестовый метод, который будет вызывать нашу форму (листинг 7.1).

Листинг 7.1

```
package ideabook.ch7;

import javax.swing.*;

public class My1 {
    private JButton cancelButton;
    private JButton loginButton;
    private JPasswordField passwordField;
    private JTextField loginField;
    private JPanel mainPanel;

    public static void main(String[] args) {
        final My1 my1 = new My1();
        final JFrame frame = new JFrame ("Test");
```

```
        frame.getContentPane().add(my1.mainPanel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

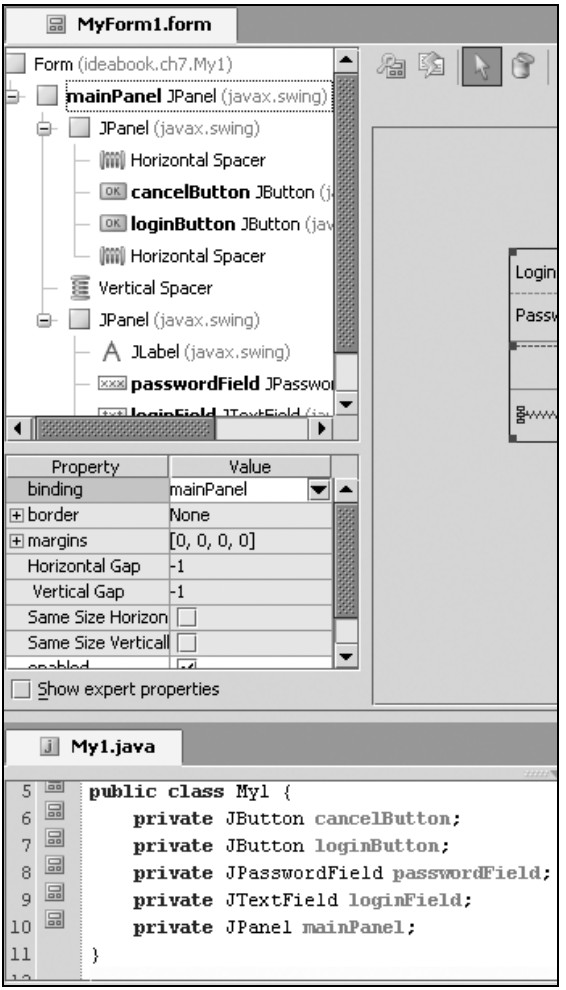


Рис. 7.26. Создание привязки к полю класса для самой верхней панели в иерархической структуре формы

Результат работы этого метода представлен на рис. 7.27.

Все работает, чего и следовало ожидать.

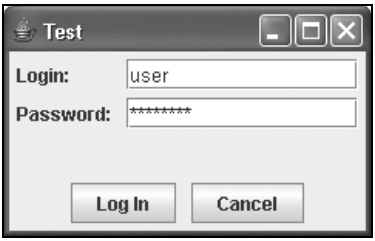


Рис. 7.27. Результат работы тестового метода класса `My1`

7.1.1.4. Как работает форма

Итак, мы создали форму и связали ее с классом. Ни в теле класса, ни где бы то ни было еще, нет кода, который создает и размещает компоненты. По крайней мере, его не видно невооруженным глазом.

Примечание

В *разделе 7.4.1* мы расскажем, как увидеть этот код.

В этом и состоит "изюминка" работы визуального редактора IntelliJ IDEA. После компиляции кода обычным компилятором Java, IntelliJ IDEA автоматически модифицирует созданный class-файл инициализацией формы. Инициализация создается автоматически на основе файла формы.

В этом, на наш взгляд, большое преимущество IntelliJ IDEA перед другими визуальными редакторами вроде Borland JBuilder, которые создают излишне объемный код. Для простых интерфейсов, состоящих из небольшого количества компонент, это, может быть, и удобно. Однако стоит начать работать со сложной формой, да еще собираемой динамически из множества компонент, как сразу расхочется пользоваться таким визуальным редактором.

С визуальным редактором IntelliJ IDEA такого не происходит, поскольку весь код дальнейшей обработки событий и данных формы пишет сам программист, а IntelliJ IDEA ему не мешает.

К моменту вызова конструктора класса, связанного с формой, все компоненты формы, соответствующие полям класса, автоматически проинициализированы и готовы к использованию. Например, так, как мы это сделали в методе `main()` из листинга 7.1, используя поле `mainPanel` класса `My1` без его явной инициализации.

7.1.2. Свойства компонент

Прежде чем продолжить работу с нашей формой, мы хотим рассказать о свойствах компонент.

Если вы обратите внимание на рис. 7.23, на нем в поле свойств компонент можно заметить, что некоторые свойства имеют серый цвет фона, а некото-

рые белый. Так сделано неспроста и связано с тем, что свойства компонент на белом фоне — это *базовые* свойства Java-объектов, соответствующие тем или иных компонентам.

Свойства на сером фоне поддерживаются самой IntelliJ IDEA. Эти свойства различаются для некоторых компонент из Swing. Дополнительные свойства — результат включения компонент в разметку формы — во взаимное размещение компонент в контейнерах. Чтобы отличать их от обычных свойств компонент, мы будем называть их *специальными*.

Компоненты, которые могут выполнять функцию контейнеров, такие как `JPanel`, `JScrollPane`, `JTabbedPane`, `JSplitPane`, даже без заданного размещения в контексте формы имеют специальные свойства. Эти свойства задают параметры размещения дочерних компонент в контейнерах.

Есть еще один тип свойств, которые в IntelliJ IDEA называются *экспертными*. В обычном режиме редактирования формы они спрятаны, но могут быть отображены, если включить соответствующую опцию в нижнем левом углу окна редактора — **Show expert properties** (Отображать экспертные свойства). Экспертные свойства относятся к базовым свойствам, т. е. они поддерживаются самими Swing-компонентами.

7.1.2.1. Свойства контейнеров

Все контейнеры (`JPanel`, `JScrollPane`, `JTabbedPane`, `JSplitPane`) поддерживают свойство **Border** (Рамка), которое имеет следующие параметры:

- **type** (тип) — тип рамки (рис. 7.28):
  - **None** (Без рамки);
  - **Bevel Lowered** (Утопленная);
  - **Bevel Raised** (Приподнятая);
  - **Etched** (Вытравленная);
- **title** (заголовок) — заголовок контейнера.

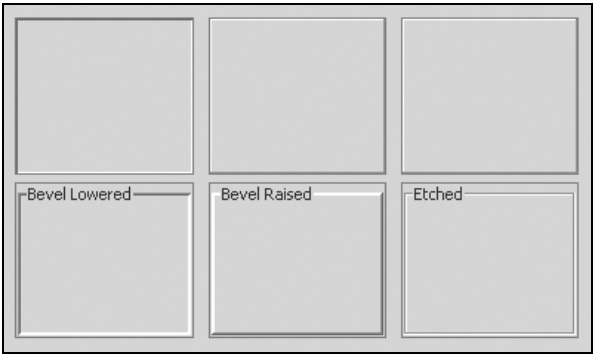


Рис. 7.28. Три типа рамок без заголовка и с заголовком

Контейнер `JPanel` используется для создания размещения компонент, поэтому у него имеются некоторые дополнительные свойства:

- **margins** (границы) — отступ от края контейнера:
  - **top** (сверху);
  - **left** (слева);
  - **bottom** (снизу);
  - **right** (справа);
- **Horizontal Gap** (Горизонтальный пробел) — горизонтальное расстояние между ячейками размещения в случае, если в контейнере размещены компоненты;
- **Vertical Gap** (Вертикальный пробел) — вертикальное расстояние между ячейками размещения в случае, если в контейнере размещены компоненты;
- **Same Size Horizontally** (Одинаковый размер по горизонтали) — возможность сохранять одинаковый размер ячеек размещения по горизонтали;
- **Same Size Vertically** (Одинаковый размер по вертикали) — возможность сохранять одинаковый размер ячеек размещения по вертикали.

Все контейнеры могут присутствовать в форме на правах компоненты, т. е. являться содержимым других контейнеров. Если контейнер расположен внутри другого контейнера, у него появляются все свойства, которые доступны для компонент. Эти свойства описаны в *разделе 7.1.2.2*.

7.1.2.2. Свойства компонент

О свойствах компонент форм мы будем говорить в предположении, что они уже размещены некоторым образом внутри контейнеров.

Все компоненты поддерживают предопределенный набор свойств:

- **binding** (привязка) — привязка к полю класса;
- **Horizontal Size Policy** (Политика горизонтального размера) — опции размера по горизонтали:
  - **Can Shrink** (Может уменьшаться);
  - **Can Grow** (Может увеличиваться);
  - **Want Grow** (Хочет увеличиваться);
- **Vertical Size Policy** (Политика вертикального размера):
  - **Can Shrink** (Может уменьшаться);
  - **Can Grow** (Может увеличиваться);

- **Want Grow** (Хочет увеличиваться);

❒ **Fill** (Заполнение):

- **None** (Отсутствует) — компонента не будет увеличивать своего размера вместе с увеличением размера контейнера;
- **Horizontal** (По горизонтали) — компонента будет увеличивать свой горизонтальный размер вместе с увеличением размера контейнера;
- **Vertical** (По вертикали) — компонента будет увеличивать свой вертикальный размер вместе с увеличением размера контейнера;
- **Both** (В обоих направлениях) — компонента будет увеличивать свой размер во всех направлениях при увеличении размеров контейнера;

❒ **anchor** (Привязка) — расположение компоненты внутри контейнера:

- **Center** (Центр) — по центру;
- **West** (Запад) — у левого края;
- **East** (Восток) — у правого края;
- **North** (Север) — у верхней границы;
- **South** (Юг) — у нижней границы;
- **North-West** (Северо-запад) — в левом верхнем углу;
- **North-East** (Северо-восток) — в правом верхнем углу;
- **South-West** (Юго-запад) — в правом нижнем углу;
- **South-East** (Юго-восток) — в левом нижнем углу;

❒ **Row Span** (Количество занимаемых ячеек сетки по горизонтали);

❒ **Column Span** (Количество занимаемых ячеек сетки по вертикали);

❒ **Minimum Size** (Минимальный размер):

- **width** (ширина) — ширина в пикселах (-1, если не определена);
- **height** (высота) — высота в пикселах (-1, если не определена);

❒ **Preferred Size** (Предпочтительный размер):

- **width** (ширина) — ширина в пикселах(-1, если не определена);
- **height** (высота) — высота в пикселах (-1, если не определена);

❒ **Maximum Size** (Максимальный размер):

- **width** (ширина) — ширина в пикселах (-1, если не определена);
- **height** (высота) — высота в пикселах (-1, если не определена).

7.1.2.3. Политика размера компонент

Компоненты могут иметь несколько свойств, от которых может зависеть их размер в форме. Во-первых, это политика размера (Size Policy). Существует восемь различных политик размера:

- **Fixed** (Фиксированная) — размер компоненты фиксирован и задается значениями **Minimum Size** (Минимальный размер) и **Preferred Size** (Предпочтительный размер);
- **Can Shrink** (Может уменьшаться) — размер компоненты может уменьшаться. Границы изменения размера задаются значениями **Minimum Size** (Минимальный размер) и **Preferred Size** (Предпочтительный размер);
- **Can Grow** (Может увеличиваться) — размер компоненты может увеличиваться. Границы изменения размера задаются значениями **Minimum Size** (Минимальный размер) или **Preferred Size** (Предпочтительный размер) и **Maximum Size** (Максимальный размер);
- **Want Grow** (Хочет увеличиваться) — размер компоненты может увеличиваться и будет увеличиваться в первую очередь при увеличении размера контейнера. Границы изменения задаются значениями **Minimum Size** (Минимальный размер) или **Preferred Size** (Предпочтительный размер) и **Maximum Size** (Максимальный размер);
- **Can Shrink, Can Grow** (Может уменьшаться и увеличиваться) — размер компоненты может уменьшаться и увеличиваться. Объединение политик размера **Can Shrink** (Может уменьшаться) и **Can Grow** (Может увеличиваться). Минимальный размер компоненты задается значение **Minimum Size** (Минимальный размер), максимальный размер компоненты задается значением **Maximum Size** (Максимальный размер), размер по умолчанию задается значением **Preferred Size** (Предпочтительный размер);
- **Can Shrink, Want Grow** (Может уменьшаться и хочет увеличиваться) — размер компоненты может уменьшаться и будет увеличиваться при увеличении размера контейнера в первую очередь. Объединение политик размера **Can Shrink** (Может уменьшаться) и **Want Grow** (Хочет увеличиваться). Минимальный размер компоненты задается значением **Minimum Size** (Минимальный размер), максимальный размер компоненты задается значением **Maximum Size** (Максимальный размер), размер по умолчанию задается значением **Preferred Size** (Предпочтительный размер);
- **Can Grow, Want Grow** (Может и хочет увеличиваться) — политика размера аналогична политике **Want Grow** (Хочет увеличиваться);
- **Can Shrink, Can Grow, Want Grow** (Может уменьшаться, может и хочет увеличиваться) — политика размера аналогична политике **Can Shrink, Want Grow** (Может уменьшаться, хочет увеличиваться).

Некоторые политики имеют схожую функциональность.

Вариантов взаимного расположения компонент с различными политиками размера может быть очень много. Так, для двух компонент таких сочетаний может быть уже 64 (8^2), хотя на практике пользоваться всеми возможными сочетаниями приходится достаточно редко.

7.1.2.4. Свойства распорок

Компоненты-распорки в IntelliJ IDEA имеют меньше свойств, чем все другие компоненты. Это связано с тем, что распорки имеют только одно измерение, т. е. они имеют либо ширину (для горизонтальных распорок), либо высоту (для вертикальных).

Поэтому у горизонтальных распорок отсутствует политика вертикальных размеров, а у вертикальных — политика горизонтальных размеров. Значения минимального, предпочтительного и максимального размера игнорируется распорками в следующих случаях: высота игнорируется горизонтальными распорками, ширина — вертикальными.

7.1.3. Размещение компонент

Прежде, чем приступить к дальнейшему усовершенствованию нашей формы, нужно рассказать о дополнительных возможностях размещения компонент.

На рис. 7.29 представлена форма с шестью компонентами, размещенными в сетке.

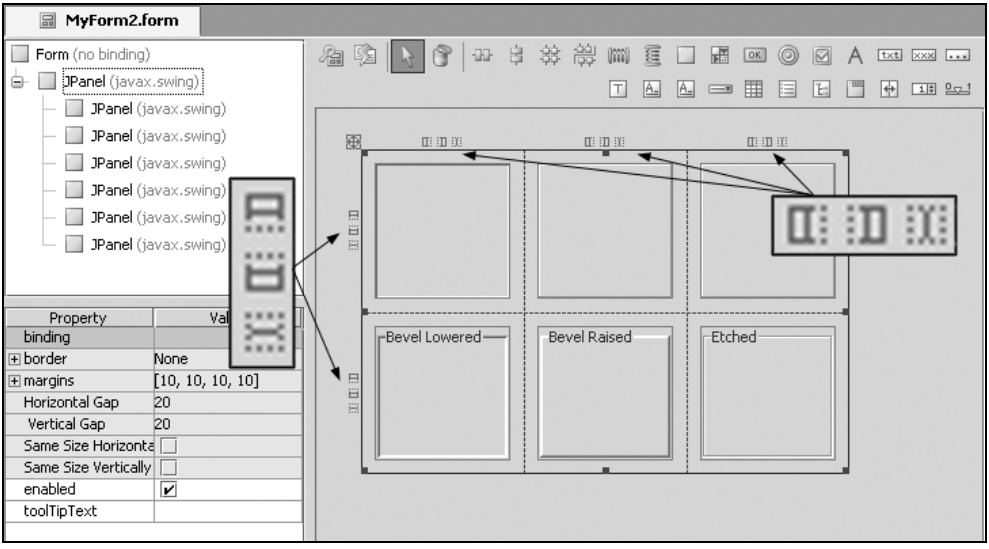


Рис. 7.29. Форма с шестью компонентами, размещенными в сетке

Если щелкнуть левой кнопкой мыши на сетке между компонентами, она будет отображена в виде пунктирных линий между компонентами. Дополнительно над каждой колонкой и строкой в сетке разметки имеются три кнопки (на рис. 7.29 они увеличены):

- Для колонок:
  - **Insert column before this** (Вставить колонку перед этой);
  - **Insert column after this** (Вставить колонку после этой);
  - **Split column** (Разделить колонку);
- Для строк:
  - **Insert row before this** (Вставить строку перед этой);
  - **Insert row after this** (Вставить строку после этой);
  - **Split row** (Разделить строку).

Давайте посмотрим на примере, как работают эти кнопки. Начнем с колонок.

7.1.3.1. *Insert column before this (Вставить колонку перед этой)*

Если нажать кнопку **Insert column before this** (Вставить колонку перед этой) для первой колонки, то первая колонка станет второй (рис. 7.30).

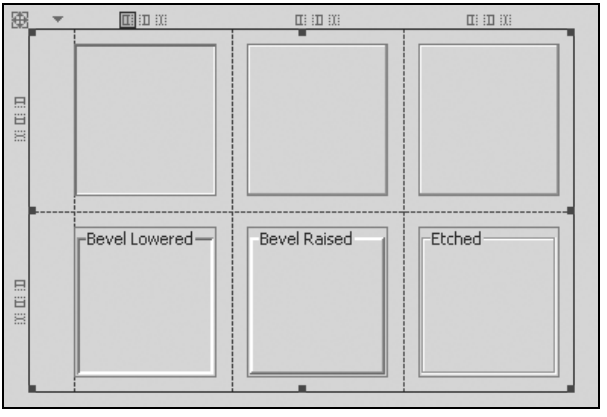


Рис. 7.30. Добавление колонки перед первой колонной

Добавилась колонка, над которой размещена кнопка в виде стрелочки. Нажатие на эту кнопку открывает контекстное меню команд для новой колонки (рис. 7.31).

В контекстном меню доступны следующие команды:

- **Insert column before this** (Вставить колонку перед этой);
- **Insert column after this** (Вставить колонку после этой);

- **Split column** (Разделить колонку);
- **Delete column** (Удалить колонку).

Первые три команды идентичны для всех колонок, а последняя — **Delete column** (Удалить колонку) — актуальна только для пустой колонки — той, в ячейках которой нет размещенных компонент.

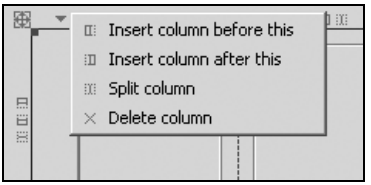


Рис. 7.31. Контекстное меню новой колонки

7.1.3.2. *Insert column after this (Вставить колонку после этой)*

Результат выполнения команды **Insert column after this** (Вставить колонку после этой) представлен на рис. 7.32.

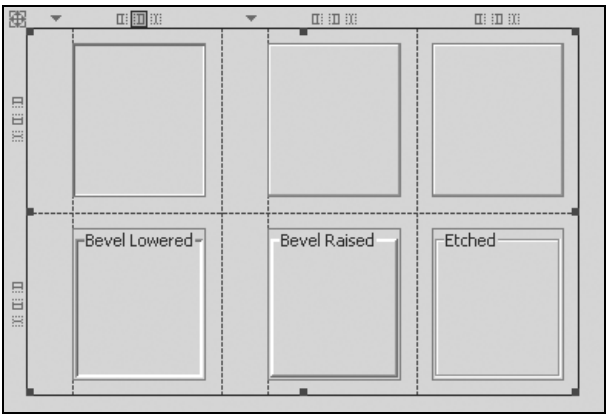


Рис. 7.32. Добавление колонки после первой колонки

7.1.3.4. *Split column (Разделить колонку)*

Команда разделения колонки несколько отличается от двух предыдущих. Если в ячейках колонки нет компонент, например, как у первой колонки на рис. 7.32, команда **Split column** (Разделить колонку) просто создаст две пустые колонки.

Если в ячейках колонки есть компоненты, результат работы команды будет несколько другим (рис. 7.33).

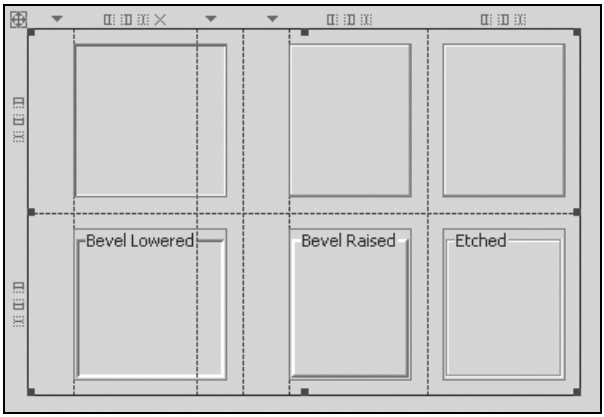


Рис. 7.33. Разделение колонки, содержащей компоненты

Колонка будет разделена на две, но и компоненты, которые были в колонке, также изменят свои свойства. Обе компоненты, размещенные в первой и второй ячейках колонки, получили значение свойства **Column Span** (Количество занимаемых ячеек сетки по вертикали) 2 (рис. 7.34). То есть компоненты просто станут шириной в две колонки.

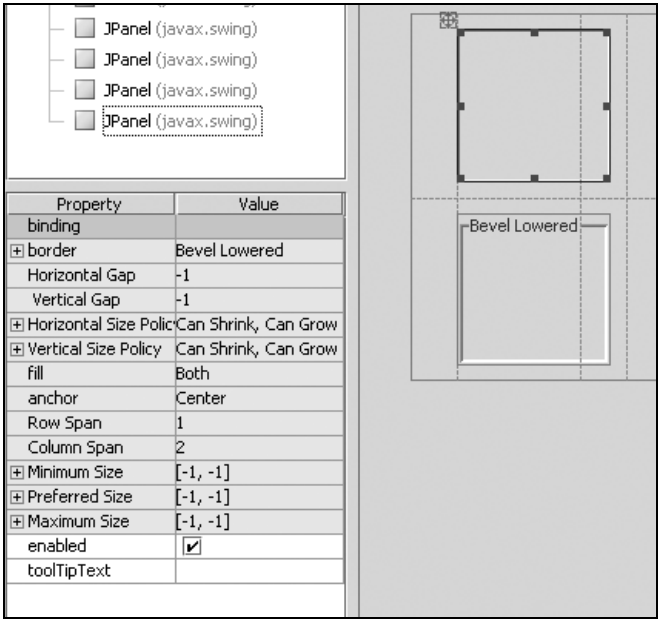


Рис. 7.34. Изменение свойства **Column Span** у компонент в разделенной колонке

7.1.3.5. *Delete column (Удалить колонку)*

Команда удаления пустой колонки возвращает разметку к первоначальному виду (см. рис. 7.32).

7.1.3.6. Перемещение компонент

В пределах разметки компоненты можно перемещать, убирая разметку и создавая ее заново в новой конфигурации, либо с помощью мыши (рис. 7.35).

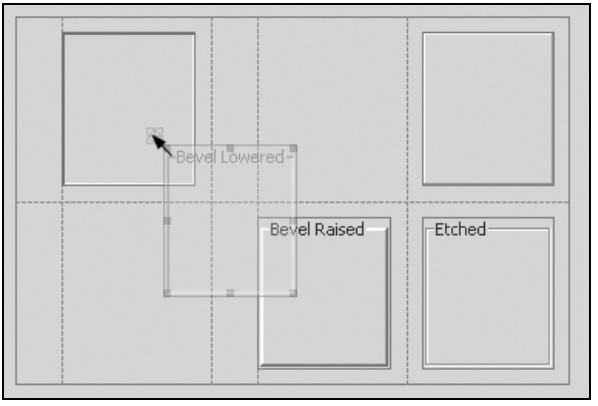


Рис. 7.35. Перемещение компонент внутри разметки

Возможно, такое обилие настроек и различных вариантов их значений может смутить, однако стоит лишь разобраться в них, как все окажется довольно простым. Благодаря возможности такой тонкой настройки свойств компонент формы, можно создавать практически любые формы пользовательского интерфейса.

7.1.3.7. Изменение размещения компонент

Если путем простого перетаскивания или добавления/удаления ячеек разметки формы невозможно добиться требуемого результата, можно убрать привязку компонент к сетке с помощью кнопки **Break Layout** (Убрать размещение), переставить компоненты так, как вам это требуется, а затем снова наложить сетку на новое размещение компонент.

Примечание

Будьте внимательны, если вы убираете размещение верхнего уровня (т. е. панель верхнего уровня, для которой у вас уже задана связка с полем класса) — информация об этой связке будет утеряна. То есть, собрав новое размещение, вам потребуется заново указать поле класса для компоненты верхнего уровня.

7.2. Создание кода

После того как форма создана, связана с классом, а ее компоненты связаны с полями класса, осталось только дописать необходимую функциональность.

Учитывая, что IntelliJ IDEA не модифицирует исходный код класса, с которым работает форма, сделать это очень просто.

Для удобства и чистоты понимания произведем с нашей формой и классом небольшой рефакторинг (*см. главу 8*), а также добавим в класс некоторую функциональность. Мы отделим функциональность панели с логином и паролем от функциональности диалога. Скопируем наши класс и форму в новые файлы, чтобы все изменения проводить в них.

1. Скопируем файл формы MyForm1.form в LoginForm.form (*см. раздел 8.1*).
2. Скопируем класс `My1` в `LoginPanel` (*см. раздел 8.1*).
3. Установим привязку формы LoginForm с классом `LoginPanel`.
4. Добавим наследование класса `LoginPanel` от класса `JPanel`.
5. Добавим конструктор без параметров, который будет содержать код, представленный в листинге 7.2.
6. Добавим в класс `LoginPanel` методы для получения логина и пароля, представленные в листинге 7.2.
7. Удалим из формы кнопки, поскольку им логичнее быть "собственностью" диалога, а не панели ввода данных. Также удалим все распорки и две нижние строки разметки, поскольку они больше не нужны (рис. 7.36).
8. И немного модифицируем метод `main()` (листинг 7.2).

Листинг 7.2

```
public LoginPanel() {
    add(mainPanel);
}

public String getLogin () {
    return loginField.getText();
}
public char [] getPassword () {
    return passwordField.getPassword();
}

public static void main(String[] args) {
```

```
        final LoginPanel loginPanel = new LoginPanel();
        final Object [] options = {"Log In", "Cancel"};
        final int result = JOptionPane.showOptionDialog(
                JOptionPane.getRootFrame(), loginPanel,
                "Log In",
                JOptionPane.YES_NO_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                null, options, options [0]);
        if (result == 0) {
            JOptionPane.showMessageDialog(JOptionPane.getRootFrame(),
                    "Log in as " + loginPanel.getLogin() + "@" +
                    new String(loginPanel.getPassword()));
        } else {
            JOptionPane.showMessageDialog(JOptionPane.getRootFrame(),
                    "Log in canceled");
        }
    }
```

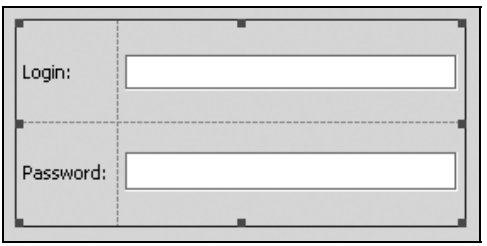


Рис. 7.36. Вид панели после удаления кнопок

Итак, мы выделили панель с полями ввода логина и пароля в отдельную компоненту. Теперь класс `LoginPanel` можно использовать в составе других форм, как любую другую Swing-компоненту.

Если компонентам в форме требуется некоторая дополнительная инициализация, например добавление Listener, объединение переключателей в `ButtonGroup` и т. п., код для инициализации нужно записать в конструкторе без параметров класса формы.

В следующем разделе мы расскажем о том, как можно добавить нашу новую компоненту в панель инструментов визуального редактора так, чтобы потом панель инструментов можно было легко включить в состав других форм.

7.3. Добавление собственных компонент

Создание интерфейса пользователя из однотипных компонент — процесс стандартный. Однако не всегда базовые компоненты поддерживают необходимую функциональность и отвечают всем требованиям. В этом случае пишутся собственные компоненты, которые затем некоторым образом добавляются в общий интерфейс.

IntelliJ IDEA позволяет легко добавлять новые компоненты в визуальный редактор, тем самым делая интуитивно ясным процесс создания сложного интерфейса пользователя из более мелких компонент.

В предыдущем разделе мы не случайно превратили форму для ввода логина и пароля в отдельную компоненту, сделав связанный с ней класс производным от `JPanel`.

Добавление собственных компонент в визуальный редактор производится с помощью диалога настройки свойств **GUI Designer** (*см. раздел 4.2*). Данный диалог (рис. 7.37) имеет две закладки:

- **General** (Общие) — общие настройки визуального редактора;
- **Palette** (Палитра) — настройки палитры компонент визуального редактора.

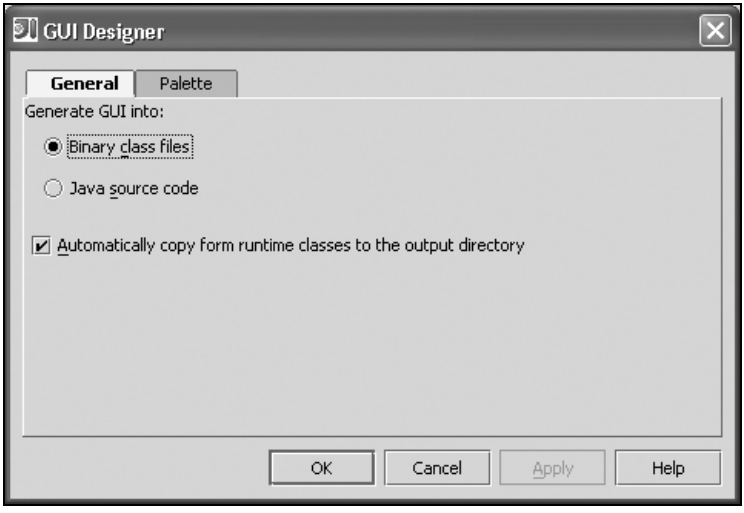


Рис. 7.37. Общий вид диалога настроек визуального редактора IntelliJ IDEA

К первой закладке **General** (Общие) мы еще вернемся в *разделе 7.4.1*, а сейчас нас интересует закладка **Palette** (Палитра) (рис. 7.38).

На закладке расположен список компонент, доступных для размещения в формах, и ряд кнопок. Компоненты сгруппированы. По умолчанию имеется лишь одна группа **Swing** (стандартные компоненты `Swing`).

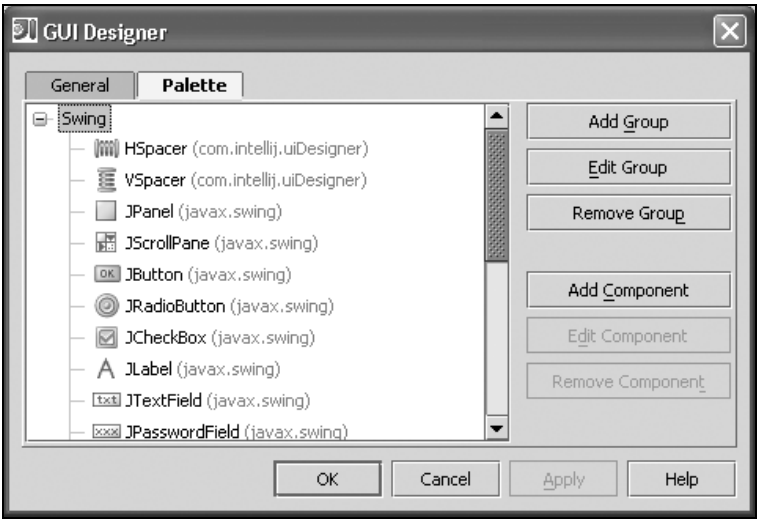


Рис. 7.38. Закладка настройки палитры компонент визуального редактора

С помощью кнопок на правой стороне закладки можно:

- **Add Group** (Добавить группу);
- **Edit Group** (Изменить название группы);
- **Remove Group** (Удалить группу);
- **Add Component** (Добавить компоненту);
- **Edit Component** (Редактировать свойства компоненты);
- **Remove Component** (Удалить компоненту).

Мы создадим новую группу My и добавим в нее нашу компоненту для ввода логина и пароля (рис. 7.39).

Рис. 7.39. Добавление компоненты `LoginPanel`

При добавлении компоненты есть возможность настроить следующие ее свойства:

- **Class** (Класс) — название класса компоненты. Класс можно выбрать с помощью диалога выбора класса, нажав на кнопку со значком "...";
- **Icon** (Пиктограмма) — пиктограмма, которая будет соотнесена с компонентой на панели инструментов визуального редактора;
- **Horizontal Size Policy** (Политика горизонтального размера) — настройки политики горизонтального размера для компоненты по умолчанию;
- **Vertical Size Policy** (Политика вертикального размера) — настройки политики вертикального размера для компоненты по умолчанию.

Когда компонента добавлена, она будет отображена в списке компонент (рис. 7.40).

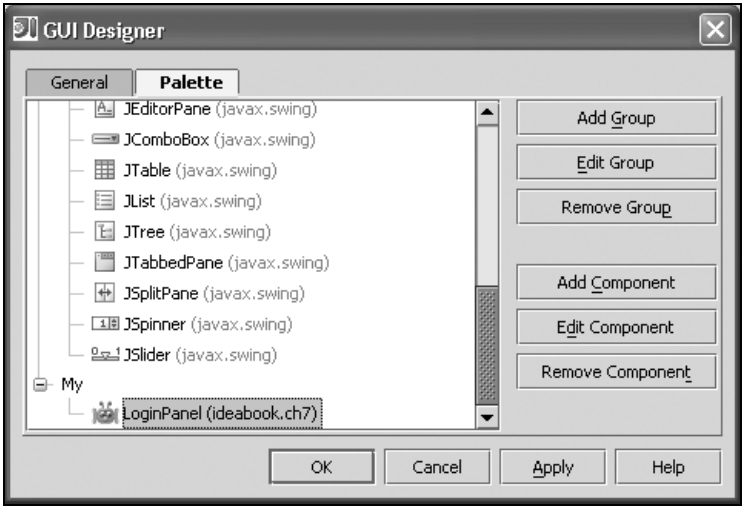


Рис. 7.40. Пользовательская компонента добавлена

IntelliJ IDEA присвоила нашей компоненте значок с симпатичным зеленым пришельцем, вместе с ним она и будет отображена на панели инструментов визуального редактора (рис. 7.41).

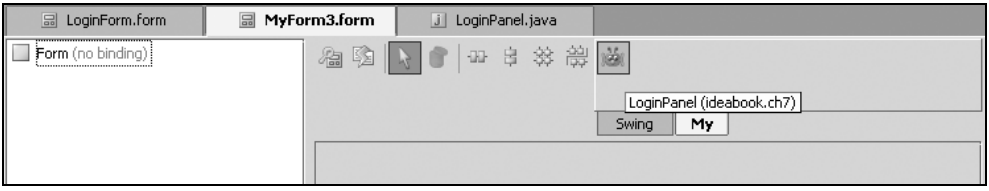


Рис. 7.41. Отображение пользовательской компоненты на панели инструментов визуального редактора

Теперь мы можем поместить нашу компоненту в любую форму так же, как мы это делали для стандартных компонент `Swing` (рис. 7.42).

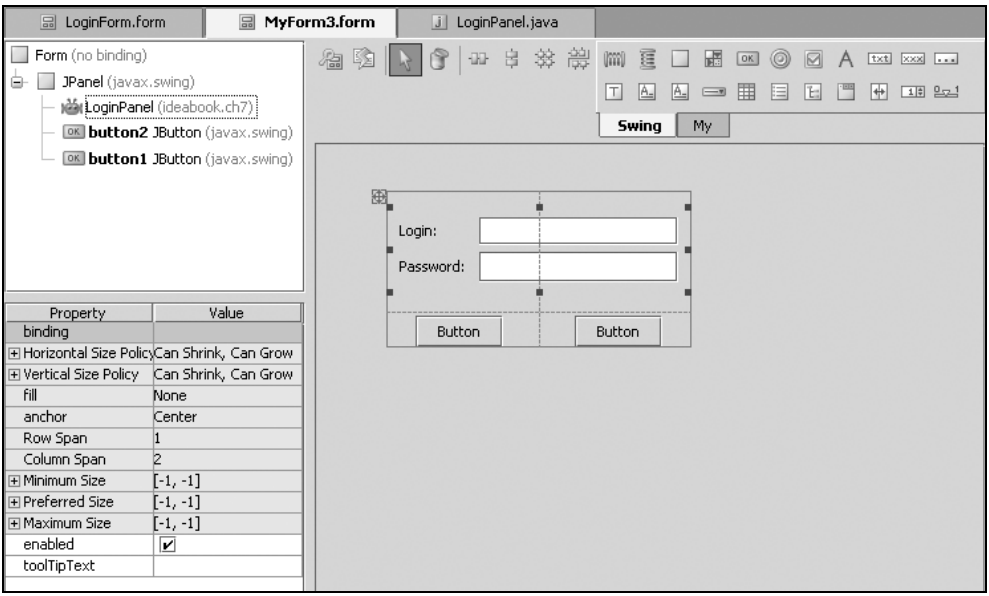


Рис. 7.42. Пользовательская компонента размещена в форме

7.4. Дополнительные возможности

Визуальный редактор IntelliJ IDEA обладает еще двумя возможностями, о которых мы не рассказали. Первая — возможность генерации исходного кода напрямую, не прибегая к модификации файлов скомпилированных классов, вторая — возможность связывать компоненты формы со свойствами `bean`-объектов с помощью мастера связей.

7.4.1. Настройка общих свойств визуального редактора

На рис. 7.37 представлен общий вид диалога настройки визуального редактора IntelliJ IDEA. Содержание второй закладки было нами рассмотрено в *разделе 7.3*, сейчас нас интересует первая закладка.

На первой закладке не так много опций:

- **Generate GUI into** (Генерировать код пользовательского интерфейса в):
  - **Binary class files** (Бинарные файлы классов). Данное значение установлено по умолчанию. IntelliJ IDEA встраивает инициализацию компонент формы прямо в файл со скомпилированным классом;

- **Java source code** (Исходный код на Java). Если установить эту опцию, IntelliJ IDEA будет создавать исходный код инициализации формы. Результат включения данной опции можно посмотреть в листинге 7.3;

❒ **Automatically copy form runtime classes to the output directory** (Автоматически копировать рабочие классы форм в выходную директорию). Данная опция позволяет автоматически копировать необходимые для работы форм классы в выходную директорию.

Листинг 7.3

```
    {
// GUI initializer generated by IntelliJ IDEA GUI Designer
// !!! IMPORTANT !!!
// DO NOT EDIT OR ADD ANY CODE HERE!
        $$$setupUI$$$();
    }

    /**
     * Method generated by IntelliJ IDEA GUI Designer
     * !!! IMPORTANT !!!
     * DO NOT edit this method OR call it in your code!
     */
    private void $$$setupUI$$$() {
        mainPanel = new JPanel();
        mainPanel.setLayout(
            new com.intellij.uiDesigner.core.GridLayoutManager(1, 1,
            new Insets(0, 0, 0, 0), -1, -1));
        final JPanel panel1 = new JPanel();
        panel1.setLayout(
            new com.intellij.uiDesigner.core.GridLayoutManager(2, 2,
            new Insets(5, 5, 5, 5), -1, -1));
        mainPanel.add(panel1,
            new com.intellij.uiDesigner.core.GridConstraints(0, 0, 1,
1, com.intellij.uiDesigner.core.GridConstraints.ANCHOR_CENTER,
com.intellij.uiDesigner.core.GridConstraints.FILL_BOTH,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_CAN_SHRINK |
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_CAN_GROW,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_CAN_SHRINK |
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_CAN_GROW,
null, null, null));
```

```
        final JLabel label1 = new JLabel();
        label1.setText("Login:");
        panel1.add(label1,
            new com.intellij.uiDesigner.core.GridConstraints(0, 0, 1,
1, com.intellij.uiDesigner.core.GridConstraints.ANCHOR_WEST,
com.intellij.uiDesigner.core.GridConstraints.FILL_NONE,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED, null,
null, null));
        loginField = new JTextField();
        loginField.setText("");
        panel1.add(loginField,
            new com.intellij.uiDesigner.core.GridConstraints(0, 1, 1,
1, com.intellij.uiDesigner.core.GridConstraints.ANCHOR_WEST,
com.intellij.uiDesigner.core.GridConstraints.FILL_HORIZONTAL,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_WANT_GROW,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED, null,
            new Dimension(150, -1), null));
        passwordField = new JPasswordField();
        panel1.add(passwordField,
            new com.intellij.uiDesigner.core.GridConstraints(1, 1, 1,
1, com.intellij.uiDesigner.core.GridConstraints.ANCHOR_WEST,
com.intellij.uiDesigner.core.GridConstraints.FILL_HORIZONTAL,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_WANT_GROW,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED, null,
            new Dimension(150, -1), null));
        final JLabel label2 = new JLabel();
        label2.setText("Password:");
        panel1.add(label2,
            new com.intellij.uiDesigner.core.GridConstraints(1, 0, 1,
1, com.intellij.uiDesigner.core.GridConstraints.ANCHOR_WEST,
com.intellij.uiDesigner.core.GridConstraints.FILL_NONE,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED,
com.intellij.uiDesigner.core.GridConstraints.SIZEPOLICY_FIXED, null,
null, null));
    }
```

Содержимое листинга 7.3 очень напоминает код, созданный Borland JBuilder, в случае использования `GribBagLayout`.

Гораздо лучше, когда IntelliJ IDEA напрямую модифицирует скомпилированные классы, потому что в исходном тексте класса отсутствует "мусор" из листинга 7.3.

7.4.2. *Data Binding Wizard (Мастер связей)*

Визуальный редактор форм IntelliJ IDEA обладает еще одной полезной возможностью связывать компоненты формы со свойствами bean-объектов. Для этого можно использовать **Data Binding Wizard** (Мастер связей) из панели инструментов визуального редактора IntelliJ IDEA.

Допустим, у нас есть форма информации о пользователе с двумя полями (рис. 7.43):

- **Name** (Имя);
- **E-mail**.

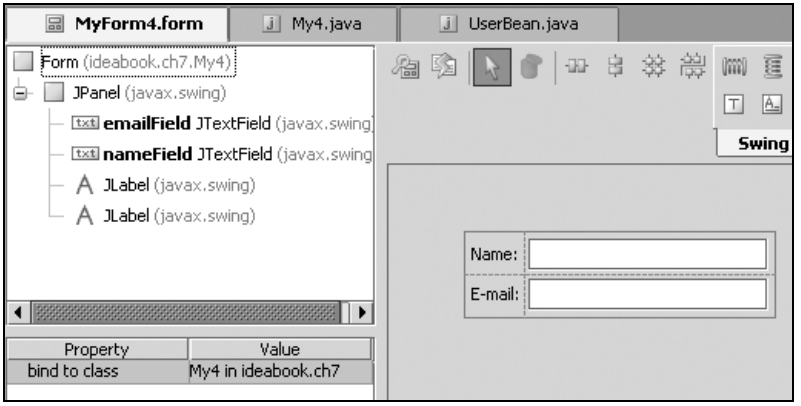


Рис. 7.43. Форма информации о пользователе

Форма MyForm4.form связана с классом `My4` (листинг 7.4).

Листинг 7.4

```
import javax.swing.*;

public class My4 {
    private JTextField nameField;
    private JTextField emailField;
}
```

Пусть также у нас есть bean-класс `UserBean`, в котором есть поля для имени и e-mail (листинг 7.5).

Листинг 7.5

```
public class UserBean {
    private String name;
    private String email;

    public UserBean() {
    }

    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

С помощью **Data Binding Wizard** (Мастер связей) мы можем связать класс формы `My4` с `bean`-классом `UserBean`. Для этого запустим командой из панели инструментов мастер связей (рис. 7.44).

Мастер связей позволяет создать `bean`-класс для формы (**Create new bean**), либо связать уже имеющийся `bean`-класс (**Bind to existing bean**). Мы выбираем вторую опцию. В поле **Class** (Класс) задаем название нашего `bean`-класса `UserBean`.

На следующем шаге мы выбираем связи между компонентами формы и полями `bean`-класса (рис. 7.45).

В нижней части текущей страницы мастера связей расположены три опции, задающие методы, которые мастер связей может создать для класса формы:

- **getData()** — метод считывает в `bean`-объект данные из формы;
- **setDate()** — метод записывает поля `bean`-объекта в форму;
- **isModified()** — метод проверяет, изменились ли данные в форме по сравнению с `bean`-объектом.

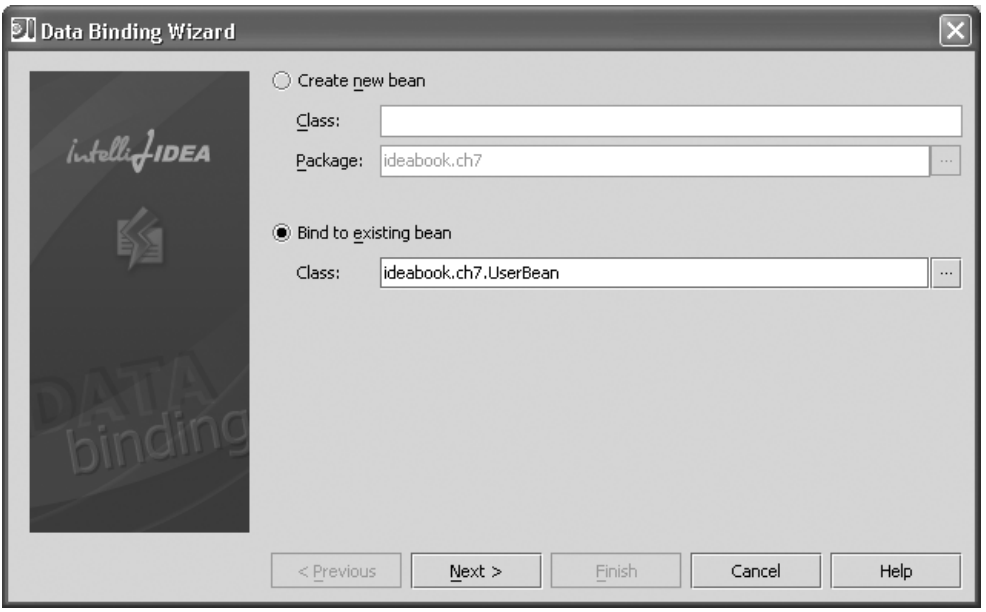


Рис. 7.44. Окно мастера связей

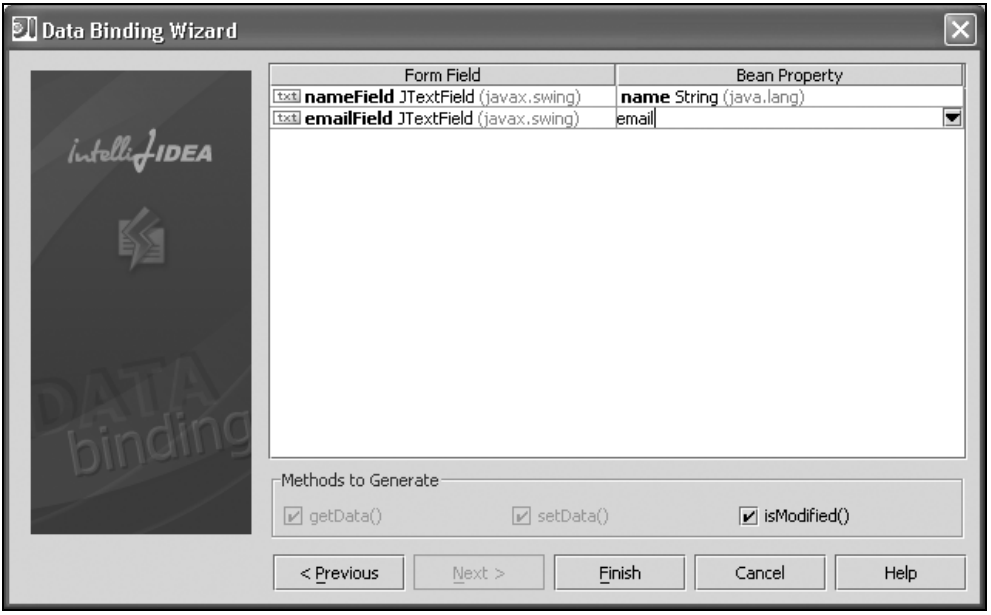


Рис. 7.45. Установка связей между компонентами формы и полями `bean`-класса

Завершив работу мастера связей кнопкой **Finish** (Завершить), мы получим код в классе формы `My4`, представленный в листинге 7.6.

Листинг 7.6

```
public void setData(UserBean data) {
    nameField.setText(data.getName());
    emailField.setText(data.getEmail());
}
public void getData(UserBean data) {
    data.setName(nameField.getText());
    data.setEmail(emailField.getText());
}

public boolean isModified(UserBean data) {
    if (nameField.getText() != null ?
        !nameField.getText().equals(data.getName()) :
        data.getName() != null)
        return true;
    if (emailField.getText() != null ?
        !emailField.getText().equals(data.getEmail()) :
        data.getEmail() != null)
        return true;

    return false;
}
```

7.4.3. Компиляция форм

В IntelliJ IDEA имеется встроенный компилятор форм. Именно этот компилятор производит модификацию классов, связанных с формами. Если проект компилируется в IntelliJ IDEA, никаких проблем с запуском не возникает. Однако очень часто проекты, особенно крупные, компилируются и собираются с помощью специальных средств сборки.

Вместе с IntelliJ IDEA поставляется библиотека для утилиты сборки Ant с компилятором форм — javac2.

Если ваш проект собирается какими-либо другими средствами, не совместимыми с Ant, придется использовать генерацию кода формы прямо в исходный текст класса (*см. раздел 7.4.1*). Также для сборки нужно будет подключить библиотеку forms_rt.jar, расположенную в директории redist внутри директории с установленной IntelliJ IDEA.

7.4.4. Интернализация (i18n)

Формы IntelliJ IDEA умеют поддерживать автоматическое связывание текстовых данных с набором ресурсов (Resource Bundle) для использования вместе с компонентами.

Рассмотрим эту возможность на примере. Мы создали форму с анкетой (рис. 7.46).

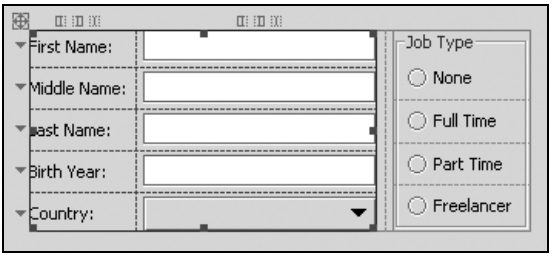


Рис. 7.46. Форма анкеты

Для текстовых постоянных полей: меток, подписей к переключателям и заголовков панели можно установить связывание с набором ресурсов. Нужно это для того, чтобы форма автоматически отображалась с использованием языка, локального для системы, на которой форма запущена.

Чтобы связать текст компоненты с набором ресурсов, сначала нужно создать этот набор — файл My5Data.properties (листинг 7.7).

Листинг 7.7

```
name.first=First Name:
name.last=Last Name:
name.mid=Middle Name:
birth.year=Birth Year:
country=Country:
job.type=Job Type:
job.type.none=None
job.type.fulltime=Full Time
job.type.parttime=Part Time
job.type.freelancer=Freelancer
```

Затем в форме для каждого текстового значения нужно задать соответствие со свойством из набора ресурсов (рис. 7.47).

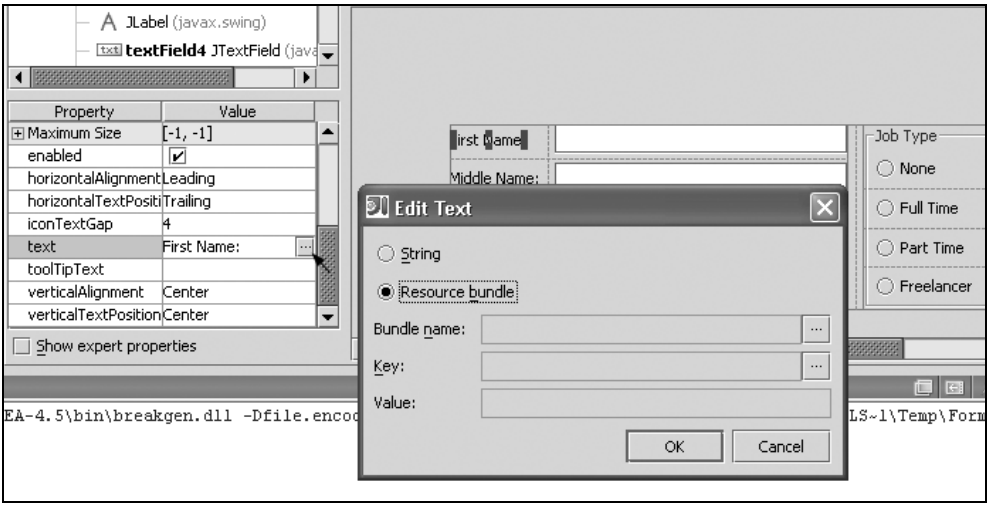


Рис. 7.47. Связывание текстового поля со свойством из набора ресурсов

Для этого нужно нажать небольшую кнопку в правой части текстового поля. В ответ на это откроется диалог (рис. 7.47), в котором следует выбрать опцию **Resource bundle** (Набор ресурсов).

Затем нужно указать три свойства:

- **Bundle name** (Название набора) — набор ресурсов можно выбрать с помощью диалога, представленного на рис. 7.48. Выбираем файл My5Data.properties;
- **Key** (Ключ ресурса) — ключ строки из файла ресурсов (рис. 7.49);
- **Value** (Значение) — строка будет выбрана автоматически (рис. 7.50).

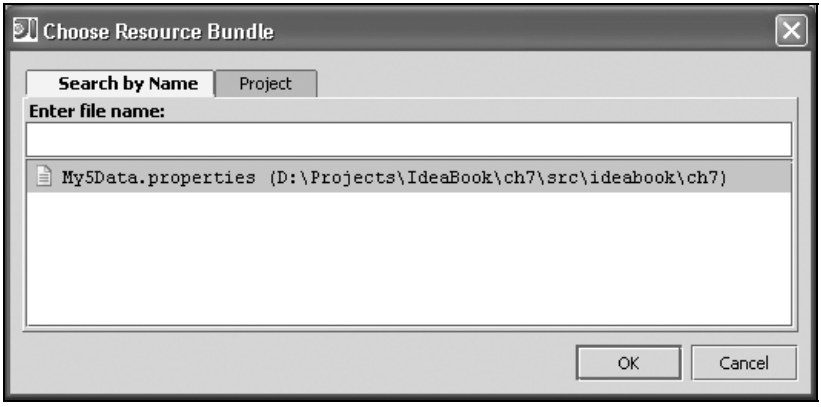


Рис. 7.48. Выбор файла с набором ресурсов

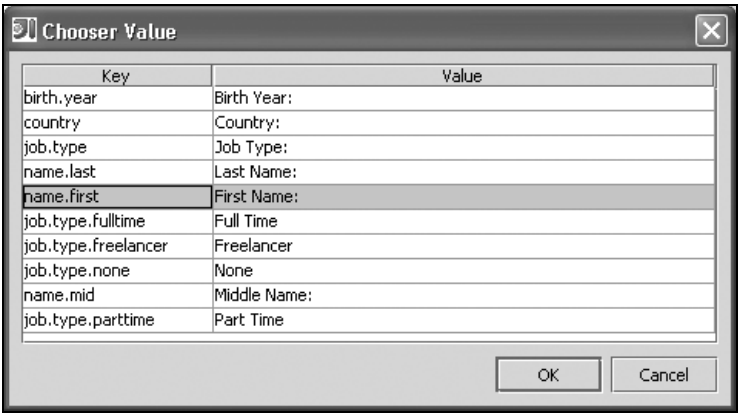


Рис. 7.49. Выбор ключа строки из файла ресурсов

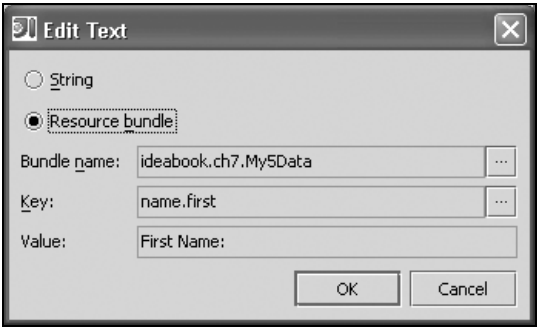


Рис. 7.50. Все значения для связывания заданы

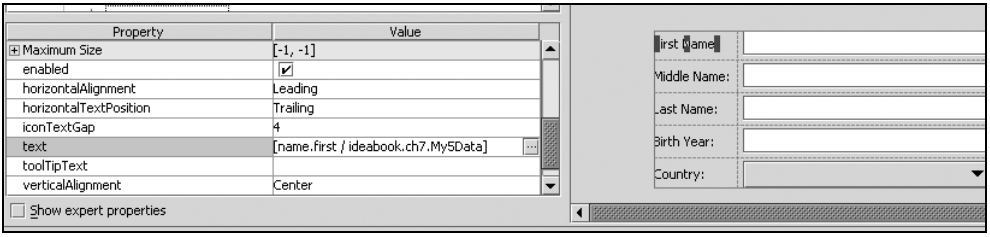


Рис. 7.51. Текст компоненты формы связан со строковым значением из набора ресурсов

После подтверждения диалога текст компоненты окажется связанным со строковым значением из набора ресурсов My5Data.properties (рис. 7.51).

Осталось связать с набором ресурсов остальные компоненты формы.

На рис. 7.52 представлен вид панели, отображенной с помощью предварительного просмотра визуального редактора IntelliJ IDEA.

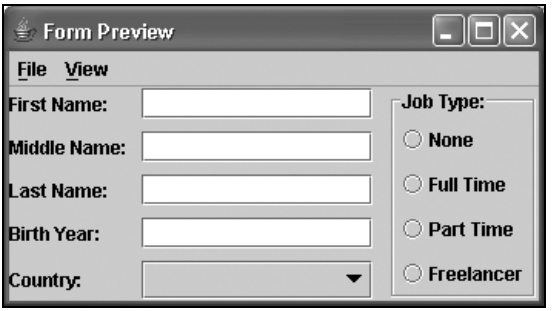


Рис. 7.52. Предварительный просмотр созданной панели

И заключительный шаг: создать наборы ресурсов для других языков, чтобы форма при запуске в системе с другой локализацией отображалась на локальном языке. Мы создадим такой набор для русского языка — My5Data_ru_RU.properties (листинг 7.8).

Листинг 7.8

```
name.first=Имя:
name.last=Фамилия:
name.mid=Отчество:
birth.year=Год рождения:
country=Страна:
job.type=Работа:
job.type.none=Безработный
job.type.fulltime=Полный день
job.type.parttime=Совместительство
job.type.freelancer=Вольнонаемный
```

Примечание

Прежде чем использовать этот набор ресурсов в нашей форме, необходимо преобразовать его к ASCII-виду, иначе русские символы в форме будут выглядеть как тарабарщина. Для этого можно использовать программу native2ascii из JDK.

В листинге 7.9 представлен файл My5Data_ru_RU.properties с русскими ресурсами, преобразованный к виду ASCII.

Листинг 7.9

```
name.first=\u0418\u043c\u044f:
name.last=\u0424\u0430\u043c\u0438\u043b\u0438\u044f:
name.mid=\u041e\u0442\u0447\u0435\u0441\u0442\u0432\u043e:
birth.year=\u0413\u043e\u0434\
\u0440\u043e\u0436\u0434\u0435\u043d\u0438\u044f:
country=\u0421\u0442\u0440\u0430\u043d\u0430:
job.type=\u0420\u0430\u0431\u043e\u0442\u0430:
job.type.none=\u0411\u0435\u0437\u0440\u0430\
\u0431\u043e\u0442\u043d\u044b\u0439
job.type.fulltime=\u041f\u043e\u043b\u043d\u044b\u0439\
\u0434\u0435\u043d\u044c
job.type.parttime=\u0421\u043e\u0432\u043c\u0435\u0441\
\u0442\u0438\u0442\u0435\u043b\u044c\u0441\u0442\u0432\u043e
job.type.freelancer=\u0412\u043e\u043b\u044c\u043d\u043e\
\u043d\u0430\u0435\u043c\u043d\u044b\u0439
```

Так как локализация нашей системы настроена на русский язык, выбрана страна — Россия (ru_RU), наша форма автоматически подхватит новый файл ресурсов и будет отображаться так, как это показано на рис. 7.53.

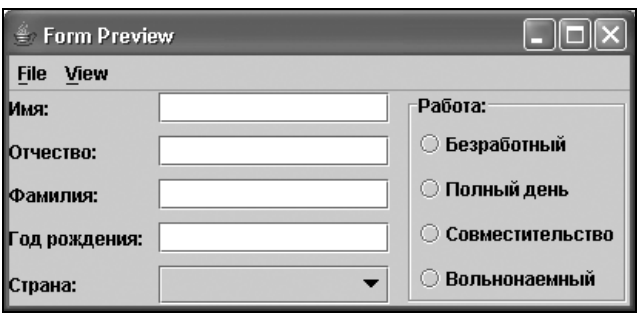
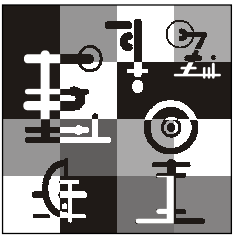


Рис. 7.53. Локализованная версия панели

Примечание

Если при отладке программы требуется проверить, как интерфейс пользователя выглядит на том или ином языке, в строке запуска можно указать JVM-параметр `user.language`, которому нужно задать значение, соответствующее требуемому языку, например, `Duser.language=en_EN` для английского языка (Великобритания). Этот параметр можно также указать и в командных параметрах IntelliJ IDEA (*см. раздел 2.4*), чтобы в визуальном редакторе для текста в компонентах по умолчанию использовался указанный язык.



Глава 8

Рефакторинг

Согласно Мартину Фаулеру (Martin Fowler), *рефакторинг* — это процесс изменения программной системы, который, не меняя внешнего поведения кода, улучшает его внутреннюю структуру. Это путь очистки кода, сводящий вероятность появления ошибок к минимуму.

IntelliJ IDEA поддерживает большое количество рефакторингов различного типа. Все команды рефакторинга собраны в главном меню в разделе **Refactor** (сокращенно от Refactoring — рефакторинг). Также команды рефакторинга доступны из контекстного меню в редакторе и в инструментальных окнах, отображающих структуру проекта или содержимое класса. Всего в IntelliJ IDEA доступно 25 различных типов рефакторинга (*см. раздел 3.2.8*). Далее мы рассмотрим каждый из них подробно.

Многие типы рефакторинга могут использовать инструментальное окно **Find** (Найти) (*см. раздел 3.4.8.4*) для вывода предварительного списка изменений, которые они предполагают произвести. Такой предварительный просмотр может быть очень полезен, поскольку иногда IntelliJ IDEA может попытаться сделать лишние изменения. В случае предварительного просмотра эти изменения можно отключить и они не будут произведены.

Команды рефакторинга перечислены в той последовательности, в которой они содержатся в разделе меню **Refactor** (Рефакторинг). Если для рефакторинга существует сочетание клавиш быстрого вызова, то это сочетание приводится в начале описания.

Многие примеры рефакторинга рассмотрены применительно к простейшим классам, но IntelliJ IDEA отлично может справляться и с более сложным кодом.

8.1. Рефакторинг *Rename (Переименовать)*

"Горячие" клавиши: <Shift>+<F6>.

Рефакторинг переименования — все вхождения данного символа заменяются его новым названием. Данный рефакторинг может выполнять переименование:

- директорий;
- файлов;
- пакетов;
- классов;
- методов;
- полей класса;
- параметров вызова метода;
- локальных переменных.

8.1.1. Переименование директорий

Для переименования директории нужно выполнить соответствующую команду, выделив требуемую директорию (рис. 8.1).

Рис. 8.1. Переименование директории

После подтверждения диалога переименования директория будет переименована (рис. 8.2).

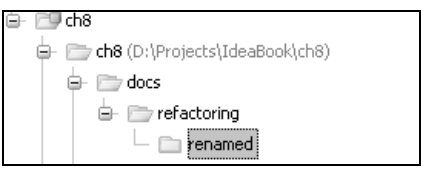


Рис. 8.2. Директория переименована

8.1.2. Переименование файлов

Переименование файла осуществляется аналогично переименованию директорий (*см. раздел 8.1.1*) (рис. 8.3).

В результате выполнения команды файл будет переименован (рис. 8.4).

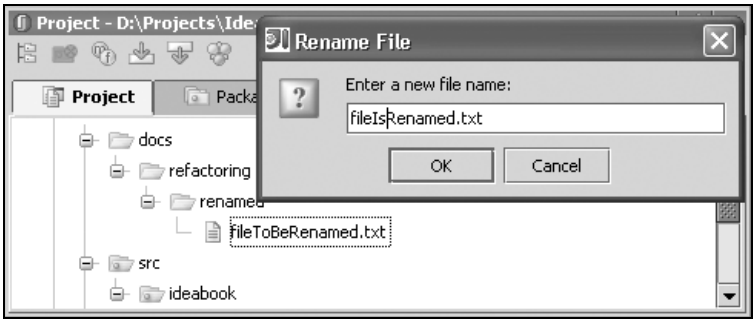


Рис. 8.3. Переименование файла

Рис. 8.4. Файл переименован

8.1.3. Переименование пакетов

Рефакторинг переименования пакетов выполняет не только перенос файлов из прежнего места в новое, но может осуществлять и замену вхождений имени пакета в строках и комментариях, а также поиск не в Java-файлах (например, в файлах комментариев JavaDoc) (рис. 8.5).

Диалог переименования пакета имеет следующие дополнительные опции:

- **Search in comments and strings** (Искать в комментариях и строках) — искать вхождения имени пакета в комментариях и строках;
- **Search in non-java files** (Искать в не Java-файлах);
- **Preview usages to be changed** (Предварительный просмотр вхождений имени пакета, которые будут изменены) — если включить данную опцию, перед тем как рефакторинг переименования пакета будет произведен, в инструментальном окне **Find** (Найти) (*см. раздел 3.4.8.4*) будут отображены все вхождения названия пакета, которые должны быть изменены (рис. 8.6).

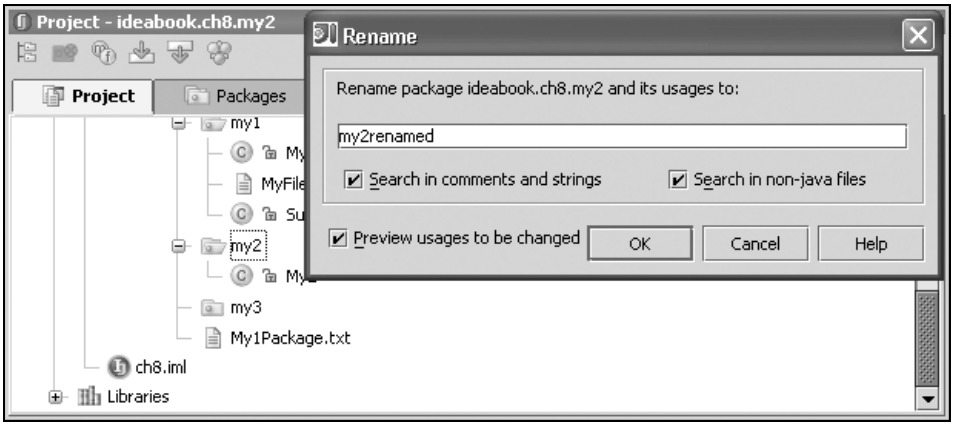


Рис. 8.5. Переименование пакетов

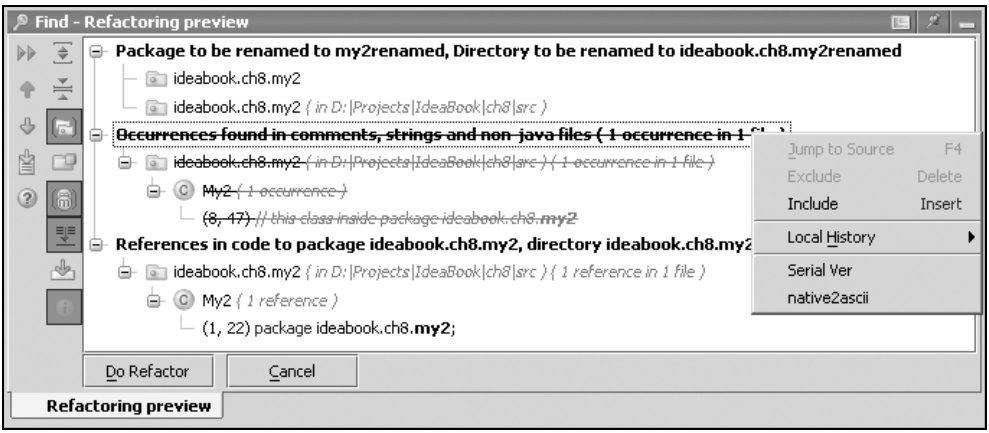


Рис. 8.6. Предварительный просмотр изменений при переименовании пакета

Мы рекомендуем всегда использовать опцию **Preview usages to be changed** (Предварительный просмотр вхождений имени пакета, которые будут изменены), поскольку часто бывает так, что IntelliJ IDEA находит лишние места в коде или в тексте файлов, которые не нужно переименовывать.

Если в результатах предварительного просмотра обнаружился лишний пункт изменений, его можно исключить с помощью команды **Exclude** (Исключить) из контекстного меню, при этом он будет выделен перечеркнутым шрифтом. Исключенный вариант можно включить обратно с помощью команды **Include** (Включить) (см. рис. 8.6).

Для подтверждения рефакторинга нужно нажать кнопку **Do Refactor** (Выполнить рефакторинг).

8.1.4. Переименование классов

Для переименования класса нужно вызвать соответствующую команду на имени класса в коде, либо в одном из инструментальных окон.

При переименовании класса, кроме его нового имени, имеется возможность задать следующие параметры:

- **Search in comments and strings** (Искать в комментариях и строках) — искать вхождения имени класса в комментариях и строках;
- **Rename variables** (Переименовывать переменные) — переименовывать переменные с типом переименовываемого класса;
- **Search in non-java files** (Искать в не Java-файлах) — искать вхождения имени класса не только в Java-файлах;
- **Rename Inheritors** (Переименовывать наследников) — переименовывать также и производные классы;
- **Preview usages to be changed** (Предварительный просмотр изменений).

8.1.5. Переименование методов

Для переименования метода нужно вызвать соответствующую команду на имени метода в коде, либо в одном из инструментальных окон (рис. 8.7).

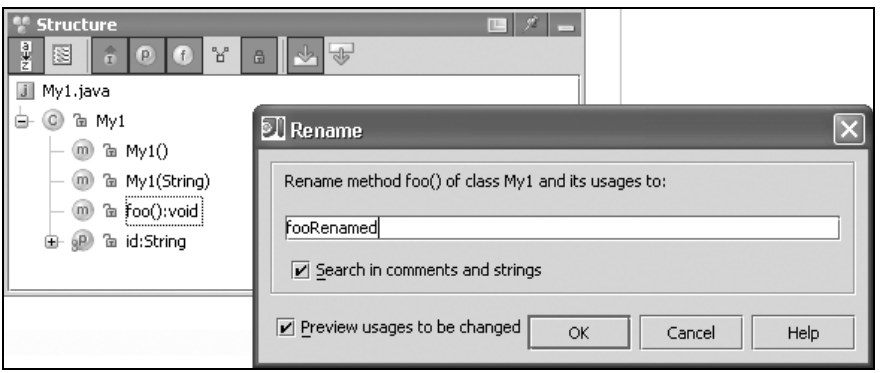


Рис. 8.7. Переименование методов

При переименовании метода имеется возможность осуществлять поиск имени метода в тексте комментариев и строках (**Search in comments and strings**).

При выполнении данного рефакторинга заменяется как само имя метода, так и имена всех переопределяющих (override) его методов в производных классах (рис. 8.8).

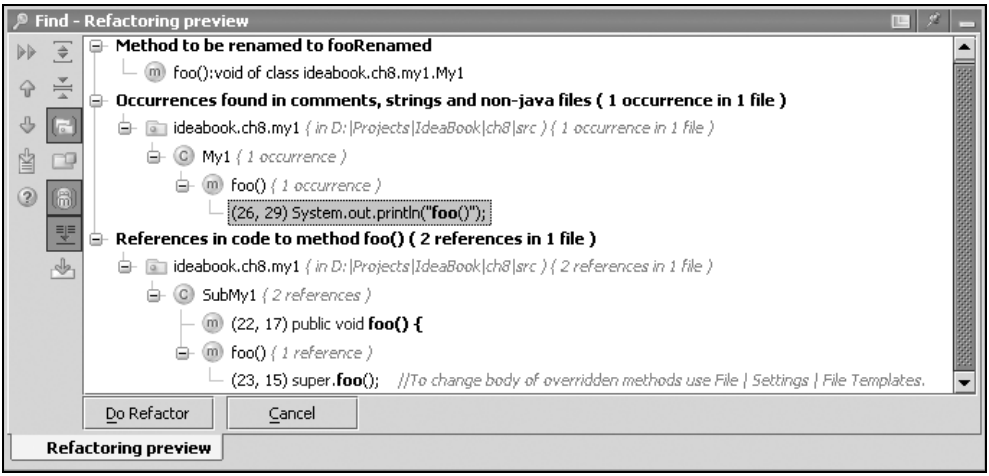


Рис. 8.8. Предварительный просмотр изменений переименования метода

8.1.6. Переименование полей класса

Для переименования поля класса необходимо выполнить команду на имени поля класса в Java-коде, либо в инструментальном окне **Structure** (Структура) (*см. раздел 3.4.8.8*) (рис. 8.9).

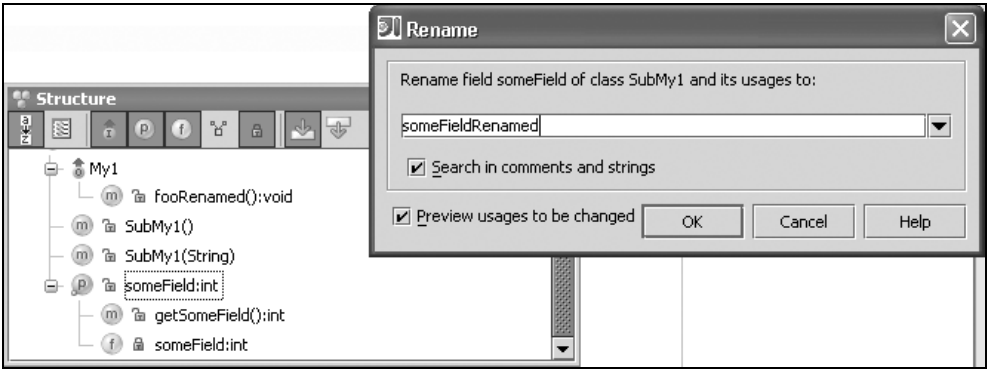


Рис. 8.9. Переименование поля класса

При переименовании поля класса имеется возможность осуществлять поиск в тексте комментариев и строках (**Search in comments and strings**).

Если у поля класса имеются методы доступа `get` или `set`, IntelliJ IDEA предложит переименовать и их в том числе (рис. 8.10).

Такая возможность автоматического переименования методов очень удобна, поскольку позволяет сохранить логическую целостность кода.

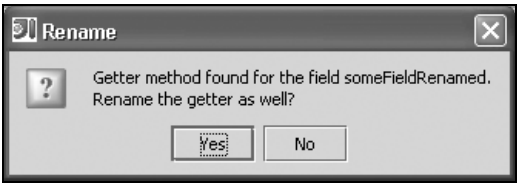


Рис 8.10. Вопрос о переименовании методов доступа `get` и/или `set`

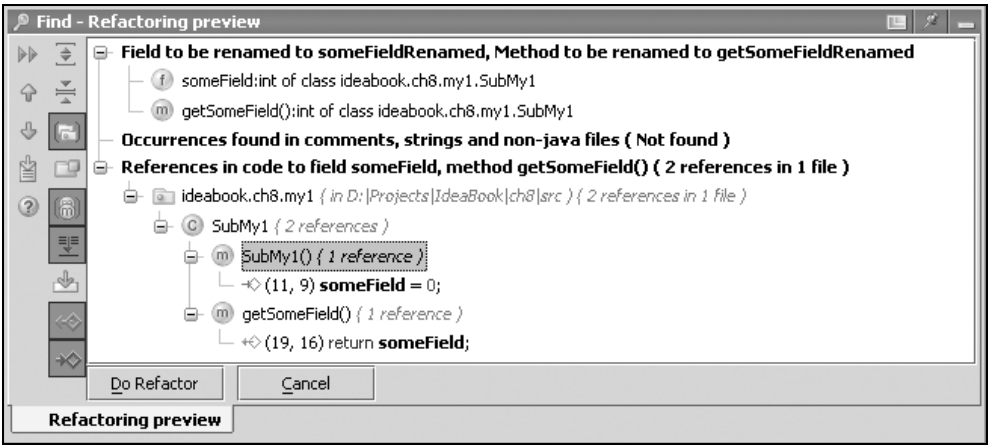


Рис. 8.11. Предварительный просмотр изменений при переименовании поля класса

Если при переименовании была установлена опция **Preview usages to be changed** (Предварительный просмотр изменений), будет отображено инструментальное окно **Find** (Найти) (рис. 8.11).

Обратите внимание на дополнительные кнопки на панели инструментов. Они позволяют исключать из просмотра найденные вхождения поля класса, замена которых зависит от типа доступа к полю класса — на чтение или на запись. Кнопки на панели инструментов аналогичны тем, что отображаются в окне **Find** (Найти) при выполнении команды **Find Usages** (Поиск обращений) (*см. раздел 3.2.3.7*).

8.1.7. Переименование параметров вызова метода

Для переименования имен параметров вызова метода необходимо вызвать команду на имени параметра (рис. 8.12).

При переименовании параметра вызова метода имеется возможность осуществлять поиск в тексте комментариев и строках (**Search in comments and strings**).

Данный рефакторинг переименовывает лишь локальное использование параметра и не меняет название параметра в переопределениях данного метода в производных классах (рис. 8.13).

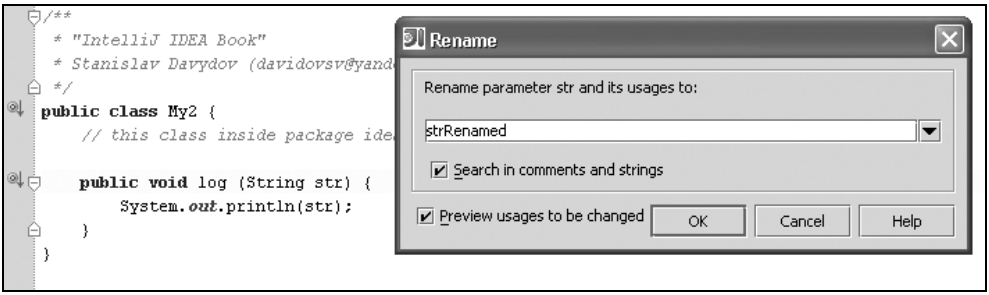


Рис. 8.12. Переименование параметра метода

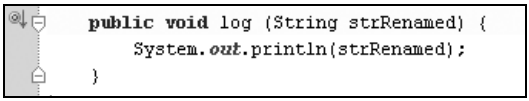


Рис. 8.13. Результат переименования параметра вызова метода

8.1.8. Переименование локальных переменных

Для переименования локальной переменной необходимо выполнить команду, установив курсор на имени локальной переменной в коде (рис. 8.14).

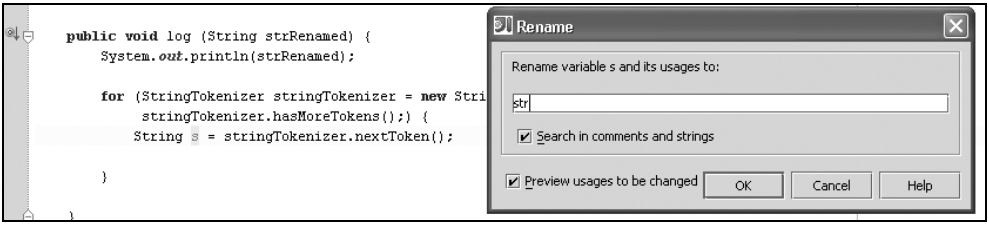


Рис. 8.14. Переименование локальной переменной

При переименовании локальной переменной имеется возможность осуществлять поиск в тексте комментариев и строках (**Search in comments and strings**).

8.2. Рефакторинг *Change Signature (Изменить сигнатуру)*

"Горячие" клавиши: <Ctrl> + <F6>.

Рефакторинг изменения сигнатуры метода, т. е. его названия, возвращаемого значения и входных параметров.

Для выполнения данного рефакторинга необходимо вызвать команду на имени метода. Расскажем о работе данной команды на примере класса из листинга 8.1.

Листинг 8.1

```
public class My2 {
    public My2() {
        try {
            FileInputStream fis = new FileInputStream("test.txt");
        } catch (FileNotFoundException e) {
            logException(e, "trying to open file for input");
        }
    }
    public void logException (Exception e, String addMessage) {
        System.err.println("Exception: ");
        e.printStackTrace(System.err);
        System.err.println("Additional message: " + addMessage);
    }
}
```

Вызовем для данного метода команду **Change Signature** (Изменить сигнатуру). Изменение сигнатуры метода осуществляется с помощью диалога, представленного на рис. 8.15.

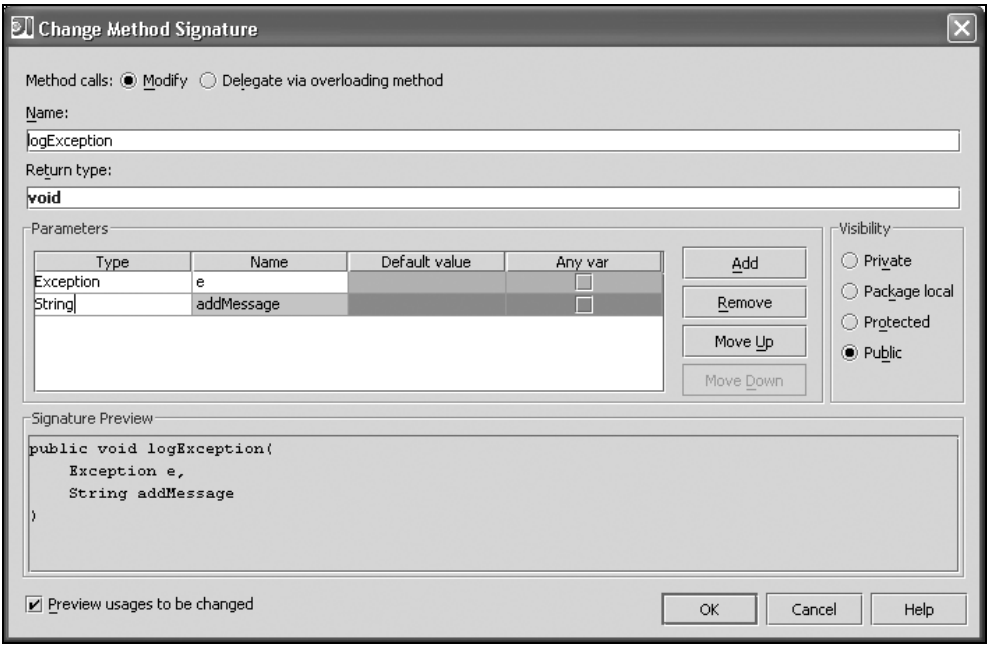


Рис. 8.15. Диалог изменения сигнатуры метода

С помощью данного диалога можно изменить:

- **Name** (Имя метода);
- **Return type** (Тип возвращаемого значения);
- **Parameters** (Число, тип и названия параметров);
- **Visibility** (Область видимости метода).

Мы внесем в наш метод следующие изменения (рис. 8.16):

1. Изменим его имя на `logThrowable`.
2. Добавим возвращаемый параметр типа `int`.
3. Изменим тип первого параметра с `Exception` на `Throwable`.
4. Добавим третий параметр `printStack` типа `boolean`.
5. И сделаем метод доступным только для пакета.

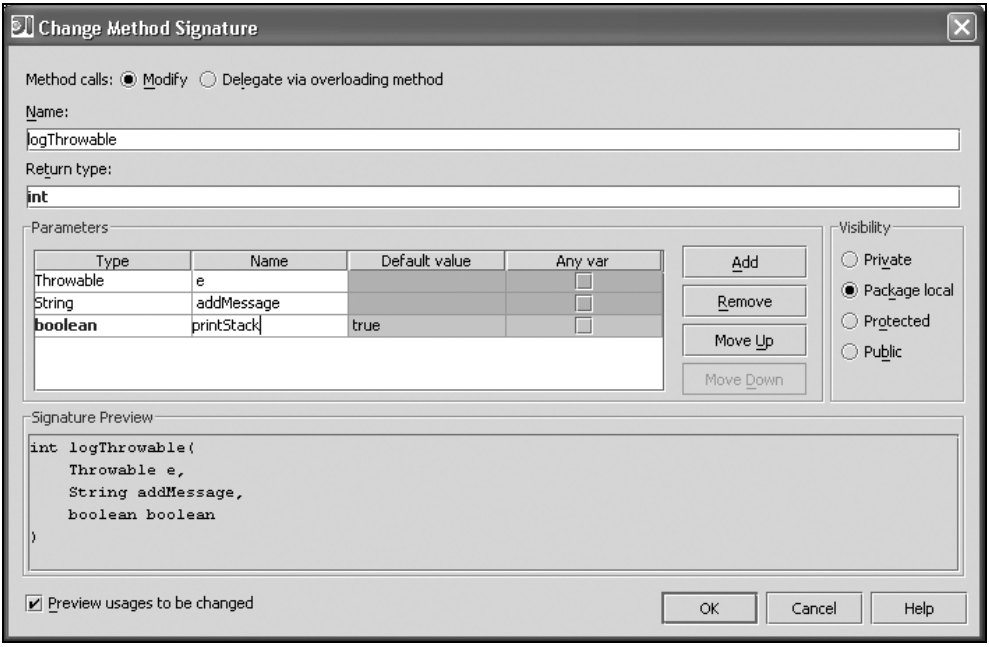


Рис. 8.16. Необходимые изменения внесены

Результат изменения сигнатуры метода можно просматривать на панели **Signature Preview** (Предварительный просмотр).

Добавление, удаление и изменение порядка параметров метода производится с помощью кнопок **Add** (Добавить), **Remove** (Удалить), **Move Up** (Переместить вверх), **Move Down** (Переместить вниз).

Данный тип рефакторинга также поддерживает возможность предварительно просматривать изменения сигнатуры метода, для этого нужно включить опцию **Preview usages to be changed** (Предварительный просмотр изменений).

Обратите внимание на колонки таблицы параметров метода, а именно на колонки **Default value** (Значение по умолчанию) и **Any** (Любой).

Для вновь добавленных параметров необходимо задать значение по умолчанию. Это значение будет добавлено в имеющиеся места вызова модифицируемого метода в качестве аргументов для новых параметров. Если это значение не задано, IntelliJ IDEA выведет сообщение об ошибке.

В колонке **Any** (Любой) находятся опции, включив которые можно указать IntelliJ IDEA самостоятельно подыскивать значения для новых переменных в имеющихся местах вызова метода. В случае, если таких переменных найдено не будет, IntelliJ IDEA подставит в вызов метода значение, заданное по умолчанию.

После подтверждения диалога IntelliJ IDEA выведет в инструментальном окне **Find** (Найти) предварительный список изменений (рис. 8.17).

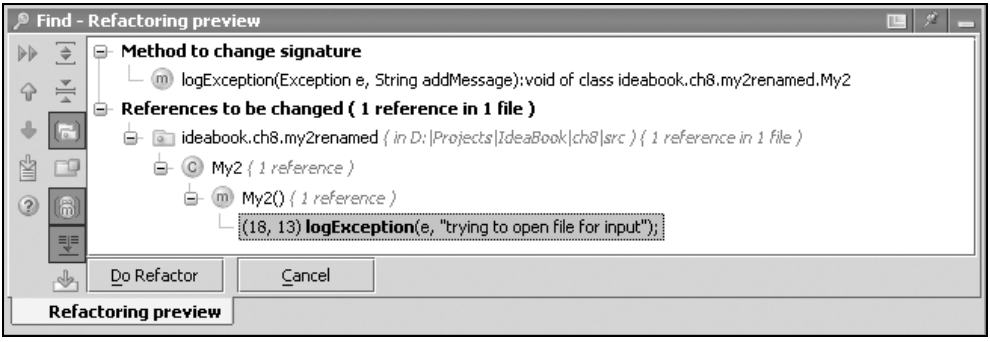


Рис. 8.17. Предварительный просмотр изменений, вызванных сменой сигнатуры метода

В результате выполнения рефакторинга, код класса изменится (листинг 8.2).

Листинг 8.2

```
public class My2 {
    public My2() {
        try {
            FileInputStream fis = new FileInputStream("test.txt");
        } catch (FileNotFoundException e) {
            logThrowable(e, "trying to open file for input", true);
        }
```

```
    }
    int logThrowable (Throwable e, String addMessage,
                      boolean printStack) {
        System.err.println("Exception: ");
        e.printStackTrace(System.err);
        System.err.println("Additional message: " + addMessage);
    }
}
```

Затем нужно добавить возвращаемое значение методу `logThrowable`, но это уже не имеет отношения к рефакторингу.

У данного рефакторинга имеется еще один режим работы, при котором вызовы метода не заменяются новыми (с новыми параметрами), а делегируются с помощью перегруженных методов. На рис. 8.18 представлен диалог вызова рефакторинга изменения сигнатуры метода `logThrowable`.

Мы вернули методу `logThrowable` первоначальный вид, удалив третий параметр, изменив тип возвращаемого значения (вернее убрали его совсем), а также сделав его доступным отовсюду (`public`).

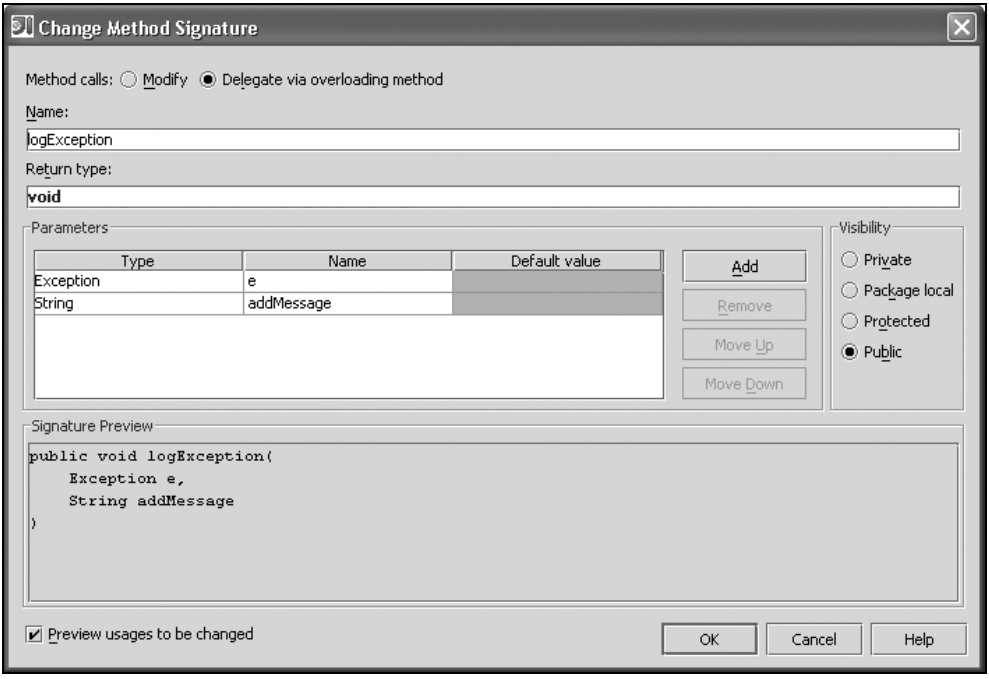


Рис. 8.18. Изменение сигнатуры метода с делегированием с помощью перегруженных методов

В результате выполнения данного рефакторинга тело метода `logThrowable` было изменено, и появился дополнительный метод `logException` (листинг 8.3).

Листинг 8.3

```
int logThrowable (Throwable e, String addMessage, boolean printStack) {
    return logException(e, addMessage);
}
public void logException (Exception e, String addMessage) {
    System.err.println("Exception: ");
    e.printStackTrace(System.err);
    System.err.println("Additional message: " + addMessage);
    return e.getStackTrace().length;
}
```

В таком виде код не будет компилироваться, но тем не менее основные изменения IntelliJ IDEA произвела сама.

8.3. Рефакторинг *Make Method Static (Сделать метод статичным)*

Рефакторинг преобразовывает обычный метод класса (instance method) в статический метод.

8.3.1. Простой случай

Данный рефакторинг имеет несколько режимов работы в зависимости от того, насколько код метода завязан на полях и методах класса. В простейшем случае, метод просто будет преобразован в статический (рис. 8.19).

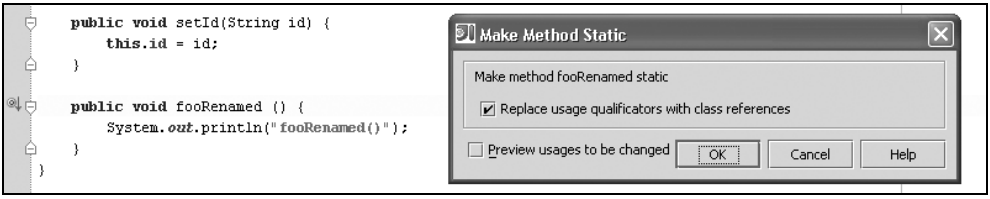


Рис. 8.19. Преобразование метода в статический

Представленный метод не имеет ссылок на поля или методы класса. Диалог преобразования метода имеет две дополнительные опции:

- **Replace usage qualificators with class references** (Заменить квалификаторы вызова метода ссылкой на класс). Если прежде вызов данного метода выглядел как `instance.fooRenamed()`, то после выполнения рефакторинга такой вызов будет заменен на `My1.fooRenamed()`, где `My1` — это имя класса, в котором определена данная функция;
- **Preview usages to be changed** (Предварительный просмотр изменений). Данная опция в нашем случае выключена по умолчанию, поскольку рефакторинг довольно прост и не предполагает возникновения спорных моментов.

В случае возникновения каких-либо проблем при выполнении рефакторинга, IntelliJ IDEA выведет соответствующее сообщение (рис. 8.20).

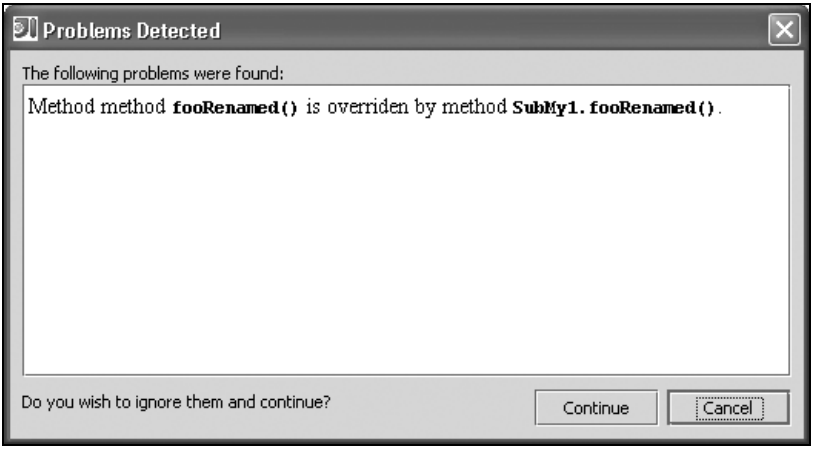


Рис. 8.20. Сообщение о проблеме при рефакторинге

В данном случае IntelliJ IDEA обнаружила, что метод, который мы собираемся сделать статическим, переопределен в другом классе.

Если проблемы не имеют большого значения, можно продолжить рефакторинг кнопкой **Continue** (Продолжить), в противном случае — отменить (**Cancel**).

Мы решили завершить рефакторинг, и метод стал статическим (рис. 8.21).

```
public static void fooRenamed () {
    System.out.println("fooRenamed()");
}
```

Рис. 8.21. Результат преобразования метода в статический

8.3.2. Сложный случай

Когда мы используем данный рефакторинг для метода, который имеет ссылки на нестатические поля и методы класса, IntelliJ IDEA вносит дополнительную функциональность в диалог проведения рефакторинга. Рассмотрим данный рефакторинг для метода `putValue()` класса `My3` из листинга 8.4.

Листинг 8.4

```
public class My3 {
    private Map map;
    public My3() {
        map = new HashMap();
    }
    public void putValue (String key, String value) {
        map.put(key, value);
    }
    public String getValue (String key) {
        return (String) map.get(key);
    }
}
```

При вызове рефакторинга **Make Method Static** (Сделать метод статическим) для метода `putValue()` будет отображен диалог, представленный на рис. 8.22.

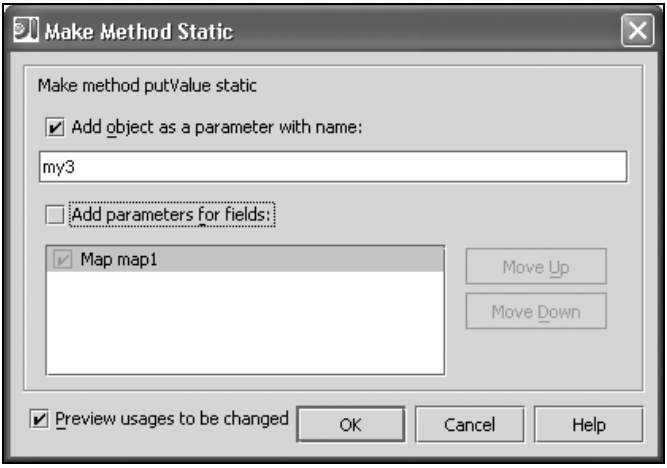


Рис. 8.22. Преобразование метода в статический

По сравнению с диалогом, представленным на рис. 8.19, здесь добавились две дополнительные опции:

- **Add object as a parameter with name** (Добавить объект, методом класса которого был метод, в качестве параметра вызова метода);
- **Add parameters for fields** (Добавить параметры вызова метода для полей класса, которые использовал метод).

В зависимости от использования метода, доступа к полям класса из производных классов и т. п. можно выбрать необходимые преобразования данного рефакторинга.

В нашем случае мы включим опцию **Add object as a parameter with name** (Добавить объект, методом класса которого был метод, в качестве параметра вызова метода) и выключим **Add parameters for fields** (Добавить параметры вызова метода для полей класса, которые использовал метод). Результат работы рефакторинга представлен в листинге 8.5.

Листинг 8.5

```
public class My3 {
    private Map map;
    public My3() {
        map = new HashMap();
    }
    public static void putValue (My3 my3, String key, String value) {
        my3.map.put(key, value);
    }
    public String getValue (String key) {
        return (String) map.get(key);
    }
}
```

8.4. Рефакторинг *Convert To Instance Method (Конвертировать в обычный метод)*

Рефакторинг преобразует статический метод в обычный метод класса (instance method).

Данный рефакторинг выполняет обратное преобразование по отношению к рефакторингу, описанному в *разделе 8.3*. Для выполнения рефакторинга необходимо, чтобы метод имел одним из параметров вызова либо экземпляр класса, для которого он определен (как в листинге 8.5), либо доступ к полям класса.

Выполним рефакторинг для метода, который мы в прошлом разделе сделали статическим.

При вызове команды для метода `putValue()` откроется диалог, представленный на рис. 8.23.

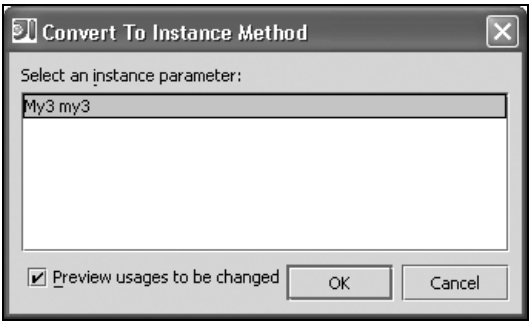


Рис. 8.23. Преобразования статического метода в обычный метод класса

IntelliJ IDEA предложит на выбор параметры статического метода, типы которых либо равны классу, где определен статический метод, либо одному из базовых классов.

В нашем случае это будет единственный параметр `my3`, являющийся экземпляром класса `My3`.

После выполнения рефакторинга метод придет к своему первоначальному состоянию (листинг 8.4).

8.5. Рефакторинг *Move (Переместить)*

"Горячая" клавиша: <F6>.

Рефакторинг перемещения сущностей. Данный рефакторинг может перемещать:

- директории;
- файлы;
- пакеты;
- классы;
- методы и поля класса.

Для выполнения перемещения очень удобно пользоваться инструментальным окном **Commander** (Коммандер) (*см. раздел 3.4.8.3*).

8.5.1. Перемещение директорий

Для перемещения директории необходимо выполнить команду **Move** (Переместить) на имени директории в одном из инструментальных окон (рис. 8.24).

Результат работы данного рефакторинга представлен на рис. 8.25.

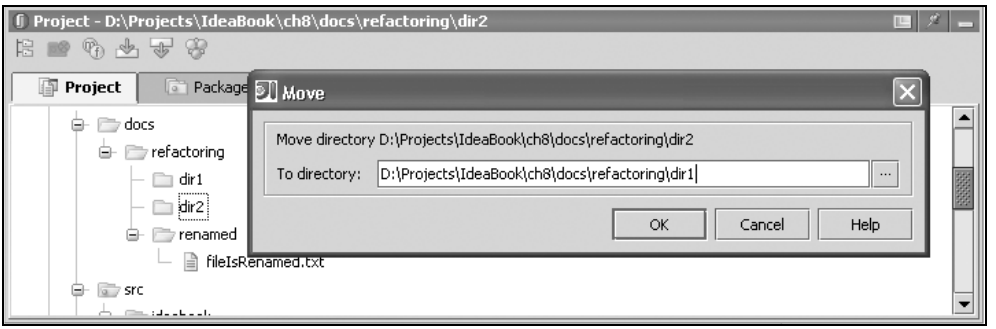


Рис. 8.24. Перемещение директории

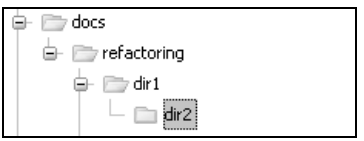


Рис. 8.25. Результат перемещения директории

8.5.2. Перемещение файлов

Для перемещения файла необходимо выполнить команду **Move** (Переместить) на файле в одном из инструментальных окон либо на файле, открытом в редакторе IntelliJ IDEA (рис. 8.26).

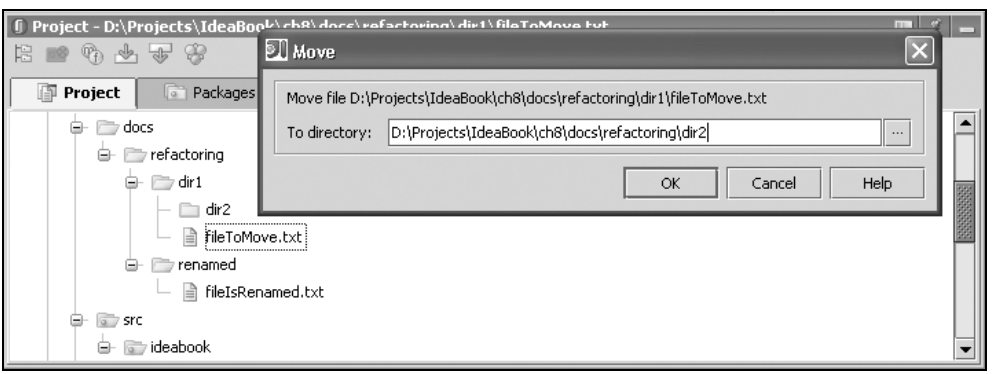


Рис. 8.26. Перемещение файла

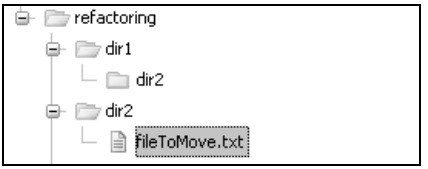


Рис. 8.27. Результат перемещения файла

При перемещении все необходимые промежуточные директории будут автоматически созданы (рис. 8.27).

8.5.3. Перемещение пакетов

Для перемещения пакета необходимо вызвать команду рефакторинга на имени пакета (рис. 8.28).

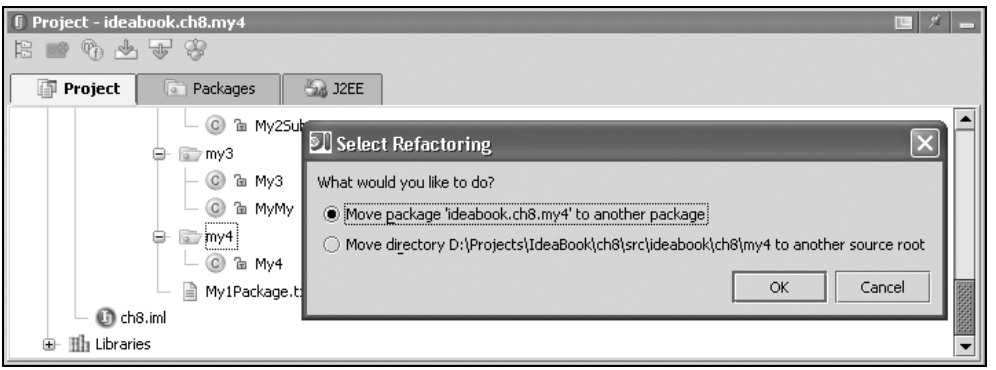


Рис. 8.28. Перемещение пакета

Если в проекте имеется несколько директорий с исходными файлами, IntelliJ IDEA предложит два варианта перемещения пакета:

- перемещение пакета в пределах одной директории с исходными файлами (**Move package... to another package**);
- перемещение директории, соответствующей выбранному пакету, в другую директорию с исходными файлами, сконфигурированную в проекте (**Move directory... to another source root**).

8.5.3.1. Перемещение пакетов в пределах одной директории с исходными текстами

При перемещении пакета в пределах одной директории с исходными текстами, IntelliJ IDEA отобразит диалог с настройкой параметров перемещения (рис. 8.29).

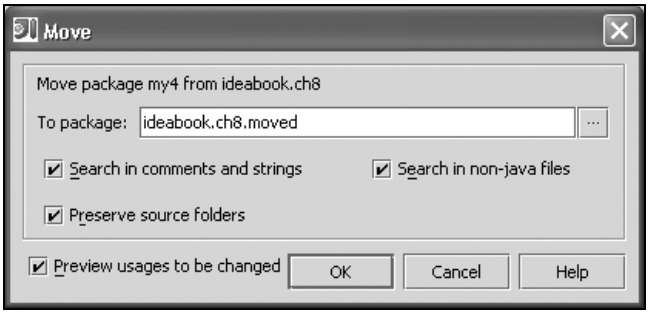


Рис. 8.29. Перемещение пакета в пределах одной директории с исходными текстами

В данном диалоге можно задать следующие опции и параметры:

- **To package** (В пакет) — место перемещения пакета;
- **Search in comments and strings** (Искать в комментариях и строках) — данная опция указывает, что будет осуществлен дополнительный поиск имени пакета в комментариях и строках;
- **Search in non-java files** (Искать не только в Java-файлах) — опция указывает на то, что использование имени папки необходимо искать не только в Java-файлах;
- **Preserve source folders** (Сохранить директорию с исходными файлами). Данная опция позволяет переместить пакет в другую директорию с исходными файлами. Об этом подробнее рассказано в *разделе 8.5.3.2*;
- **Preview usages to be changed** (Предварительный просмотр изменений).

Если опция предварительного просмотра изменений была включена, IntelliJ IDEA отобразит предполагаемые изменения в инструментальном окне **Find** (Найти) (рис. 8.30).

В результате выполнения рефакторинга пакет вместе со всем содержимым будет перенесен на новое место (рис. 8.31).

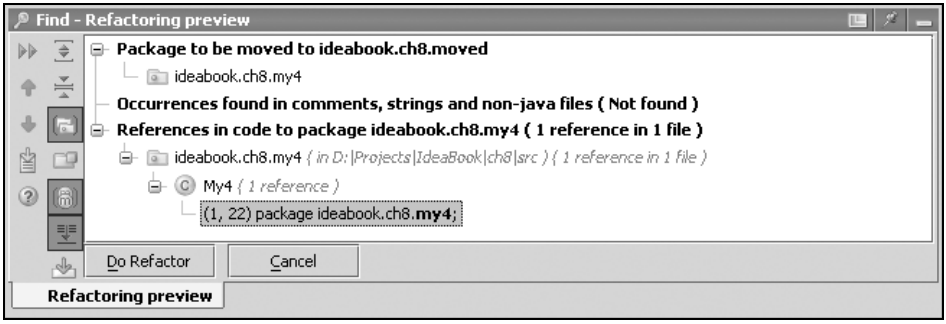


Рис. 8.30. Предварительный просмотр изменений при перемещении папки

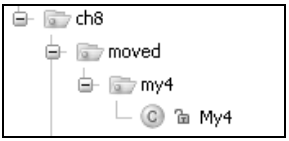


Рис. 8.31. Результат перемещения пакета

8.5.3.2. Перемещение пакетов в другую директорию с исходными текстами

В случае перемещения пакета в другую директорию с исходными текстами, либо если была отключена опция **Preserve source folders** (Сохранить директорию с исходными файлами) (*см. раздел 8.5.3.1*), будет открыт диалог выбора директории с исходными файлами для перемещения (рис. 8.32).

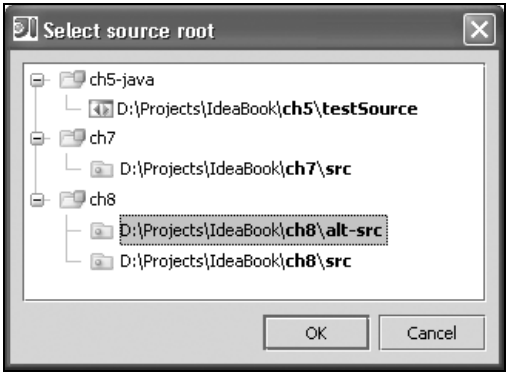


Рис. 8.32. Выбор директории с исходными файлами для перемещения

8.5.4. Перемещение классов

IntelliJ IDEA поддерживает перемещение обычных классов верхнего уровня, а также внутренних классов.

8.5.4.1. Классы верхнего уровня

Для перемещения класса необходимо вызвать данный рефакторинг на имени класса в редакторе IntelliJ IDEA или в одном из инструментальных окон (рис. 8.33).

При перемещении класса можно задать следующие параметры и опции:

- **To package** (В пакет) — переместить класс в указанный пакет;
- **Search in comments and strings** (Искать в комментариях и строках) — данная опция указывает, что будет осуществлен дополнительный поиск имени класса в комментариях и строках;

- **Search in non-java files** (Искать не только в Java-файлах) — опция указывает на то, что использование имени пакета необходимо искать не только в Java-файлах;
- **Preserve source folders** (Сохранить директорию с исходными файлами). Данная опция позволяет переместить класс в другую директорию с исходными файлами;
- **Preview usages to be changed** (Предварительный просмотр изменений).

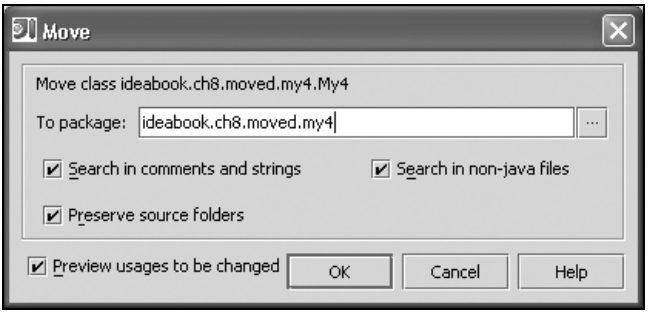


Рис. 8.33. Перемещение класса

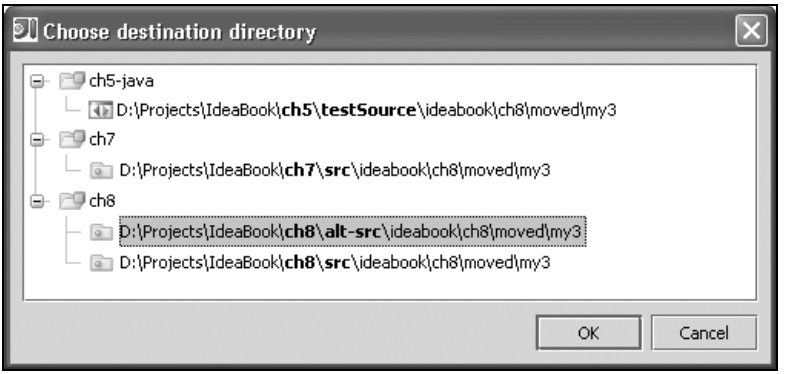


Рис. 8.34. Выбор директории с исходными файлами для перемещения класса

Если была выключена опция **Preserve source folders** (Сохранить директорию с исходными файлами), IntelliJ IDEA отобразит дополнительный диалог выбора директории с исходными файлами (рис. 8.34).

После подтверждения рефакторинга класс будет перемещен в указанное место.

8.5.4.2. Перемещение внутренних классов

Если выполнить команду **Move** (Переместить) на внутреннем классе, будет активизирована команда **Move Inner to Upper** (Переместить внутренний

класс на уровень выше). Если внутренний класс является статическим, для него также будет доступна возможность перемещения в другой класс (*см. раздел 8.5.5.2*).

При перемещении обычного внутреннего класса откроется диалог с параметрами перемещения (рис. 8.35).

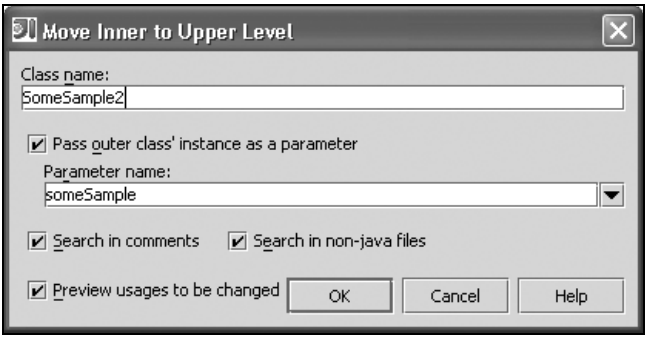


Рис. 8.35. Перемещение внутреннего класса

Доступны следующие параметры:

- **Class name** (Имя класса) — имя, на которое должен сменить свое имя внутренний класс;
- **Pass outer class' instance as a parameter** (Передавать ссылку на экземпляр внешнего класса в качестве параметра). Имя параметра задается полем **Parameter name** (Имя параметра);
- **Search in comments** (Искать в комментариях) — опция позволяет включить поиск упоминаний имени внутреннего класса в комментариях;
- **Search in non-java files** (Искать не только в Java-файлах) — опция указывает на то, что использование имени пакета необходимо искать не только в Java-файлах;
- **Preview usages to be changed** (Предварительный просмотр изменений).

Рассмотрим выполнение данной команды на примере класса из листинга 8.6.

Листинг 8.6

```
public class SomeSample {
    public void foo () {
    }

    public class SomeSampleInner {
        public void innerFoo() {
```

```
            foo();
        }
    }
}
```

После выполнения команды перемещения класс `SomeSampleInner` станет классом верхнего уровня (листинг 8.7).

Листинг 8.7

```
public class SomeSample {
    public void foo () {
    }
}

public class SomeSample2 {
    private SomeSample someSample;

    public SomeSample2(SomeSample someSample) {
        this.someSample = someSample;
    }

    public void innerFoo() {
        someSample.foo();
    }
}
```

8.5.5. Перемещение членов класса

Перемещение разрешено только для статических методов и полей класса.

8.5.5.1. Перемещение полей и методов класса

При выполнении данного рефакторинга будет отображен диалог с выбором настроек и опций перемещения, представленный на рис. 8.36.

В данном диалоге имеется возможность указать класс, в который будут перемещены методы и поля, сами методы и поля класса для перемещения (**Members to be moved**), а также область видимости (**Visibility**), с которой методы и поля будут созданы на новом месте.

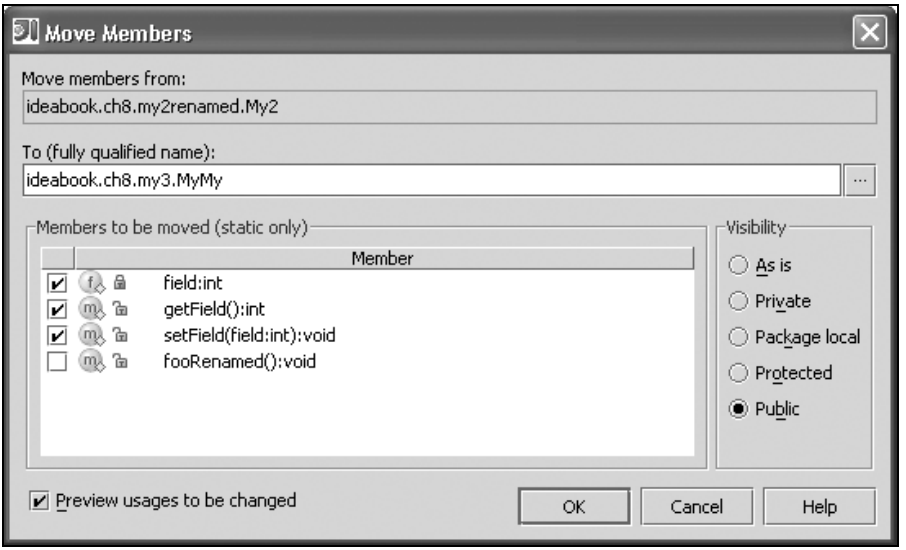


Рис. 8.36. Настройка параметров перемещения статического метода

8.5.5.2. Перемещение статических внутренних классов

В случае перемещения внутреннего статического класса имеется возможность выбрать два варианта перемещения (рис. 8.37):

- переместить внутренний класс на уровень выше;
- переместить внутренний класс в другой класс.

Рис. 8.37. Варианты перемещения внутреннего статического класса

В первом случае выполняется рефакторинг, описанный в *разделе 8.5.4.2*, во втором случае — внутренний класс переносится в другой класс как обычный статический член класса (*см. раздел 8.5.5.1*).

8.6. Рефакторинг *Copy (Копировать)*

"Горячая" клавиша: <F5>.

Рефакторинг копирования сущностей. С помощью данного рефакторинга можно копировать:

- директории;
- файлы;
- папки;
- классы.

Для выполнения копирования очень удобно пользоваться инструментальным окном **Commander** (Коммандер) (*см. раздел 3.4.8.3*).

8.6.1. Копирование директорий

При копировании директории имеется возможность указать новое имя директории, а также директорию размещения (рис. 8.38).

В результате выполнения рефакторинга будет скопирована директория и все ее содержимое (рис. 8.39).

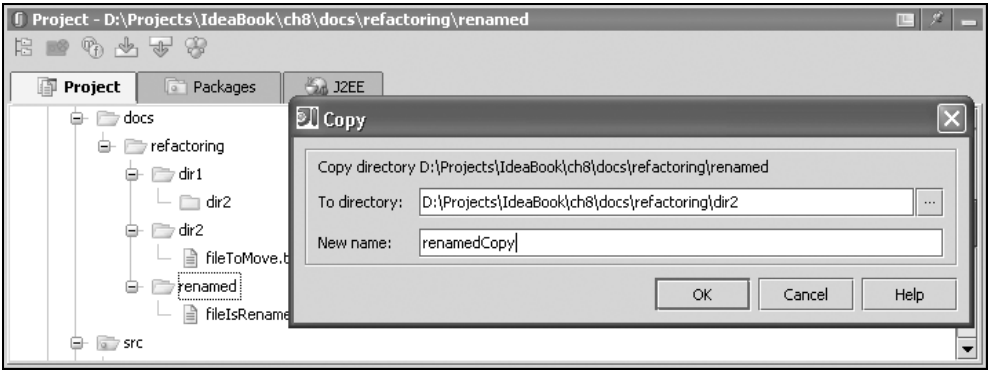


Рис. 8.38. Копирование директории

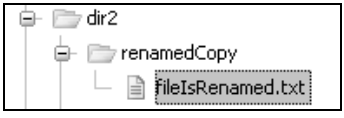


Рис. 8.39. Результат копирования директории

8.6.2. Копирование файлов

При копировании файлов можно указать новое имя файла, а также директорию, в которую он будет скопирован (рис. 8.40).

В результате выполнения рефакторинга файл будет скопирован (рис. 8.41).

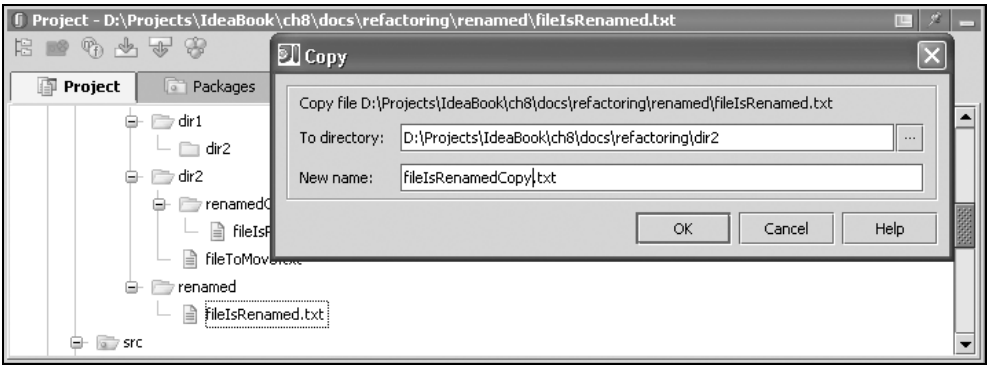


Рис. 8.40. Копирование файла

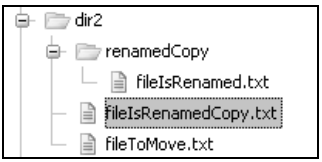


Рис. 8.41. Результат копирования файла

8.6.3. Копирование классов

При копировании класса имеется возможность указать новое имя класса, а также пакет, в который класс будет скопирован (рис. 8.42).

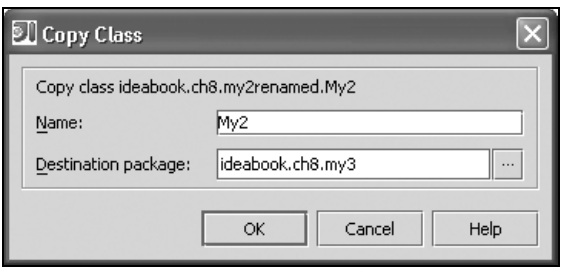


Рис. 8.42. Копирование класса

8.7. Рефакторинг *Safe Delete (Безопасное удаление)*

"Горячие" клавиши: <Alt>+<Delete>.

Рефакторинг позволяет удалить некоторую сущность, предварительно проверив, не повредит ли такое удаление остальному коду. Рефакторинг может быть выполнен:

- для класса;
- для метода;
- для поля класса.

8.7.1. Удаление класса

Для безопасного удаления класса необходимо выполнить данный рефакторинг на имени класса в редакторе или в одном из инструментальных окон (рис. 8.43).

При безопасном удалении класса имеется возможность проверить наличие ссылок на удаляемый класс не только в Java-коде, но и в комментариях и строках (**Search in comments and strings**), а также в прочих файлах (**Search in non-java files**).

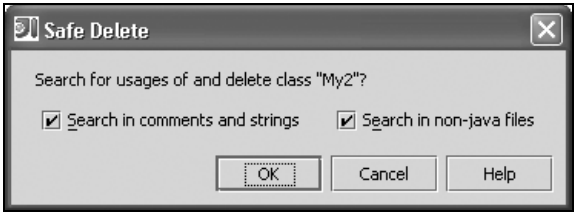


Рис. 8.43. Безопасное удаление класса

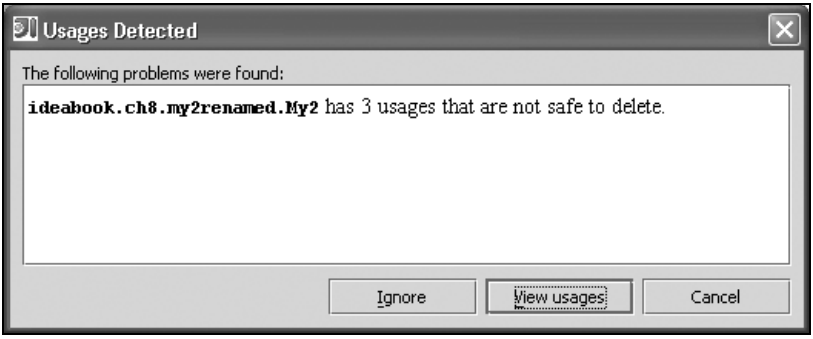


Рис. 8.44. Предупреждение об использовании класса

Если класс нигде не используется, он будет удален, в противном случае IntelliJ IDEA отобразит диалог с предупреждением (рис. 8.44).

С помощью кнопки **View usages** (Просмотр использований) можно уточнить, в каких местах используется класс. Нажав кнопку **Ignore** (Игнорировать), можно принудительно удалить класс, несмотря на возможные проблемы с целостностью кода.

8.7.2. Удаление метода

При удалении метода также имеется возможность проверять его использование не только в Java-коде, но и в комментариях и строках (рис. 8.45).

Рис. 8.45. Безопасное удаление метода

Если удаляемый метод имеет переопределяющие методы в производных классах, IntelliJ IDEA выдаст предупреждающий диалог со списком переопределений метода (рис. 8.46).

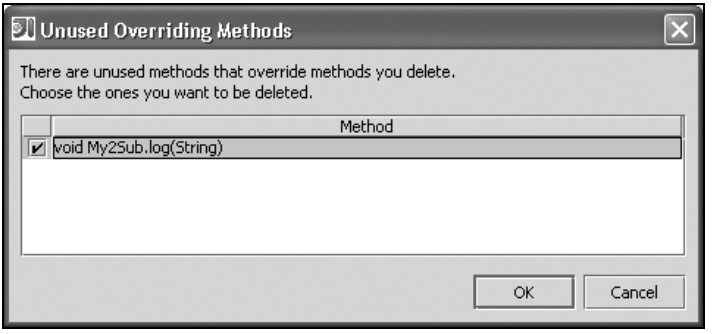


Рис. 8.46. Предупреждение о переопределяющих методах в производных классах

Если рефакторинг найдет и другие типы использования класса, он выведет предупреждение, аналогичное тому, что представлено на рис. 8.44.

8.7.3. Удаление поля класса

При удалении поля класса (рис. 8.47) рефакторинг также произведет поиск методов `get` и `set` для данного поля. Если таковые будут найдены, IntelliJ IDEA предложит произвести безопасное удаление и для них (рис. 8.48).

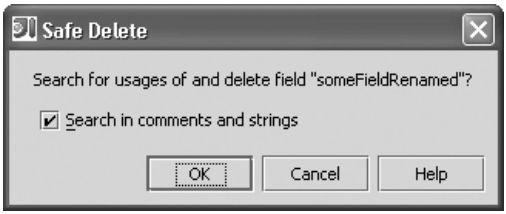


Рис. 8.47. Безопасное удаление поля класса

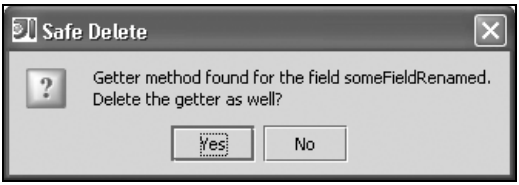


Рис. 8.48. Безопасное удаление методов `get` и `set` поля класса

Если использований поля не найдено, оно будет безопасно удалено, в противном случае будет отображено предупреждение, аналогичное представленному на рис. 8.44.

8.8. Рефакторинг *Extract Method (Выделить метод)*

"Горячие" клавиши: <Ctrl>+<Alt>+<M>.

Данный рефакторинг помогает преобразовать некоторую последовательность кода в вызов метода. Рассмотрим его работу на примере класса из листинга 8.8.

Листинг 8.8

```
public class WebColorer {
    private static int [] WEB_COLORS = {
        0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF};

    public static int webColor (int color) {
        int colorIdx;
        for (colorIdx = 0; colorIdx < WEB_COLORS.length; colorIdx++)
            if (color == WEB_COLORS [colorIdx])
                return color;
```

```
            else if (color > WEB_COLORS[colorIdx] &&
                    color < WEB_COLORS [colorIdx+1])
                break;
        int before = color - WEB_COLORS [colorIdx];
        int after = WEB_COLORS [colorIdx + 1] - color;

        if (before < after)
            return WEB_COLORS [colorIdx];
        else
            return WEB_COLORS [colorIdx + 1];
    }
}
```

Данный класс содержит метод `webColor()`, который позволяет для заданной цветовой компоненты определить ближайший цвет, который является "безопасным" цветом для Web. Составляющие "безопасных" цветов кратны `0x33`.

Используем рефакторинг выделения метода для последнего условного оператора `if-else` и двух строк перед ним с определением переменных `before` и `after` (рис. 8.49). Для этого необходимо выделить область кода, которую мы хотим перенести в метод, а затем выполнить команду рефакторинга.

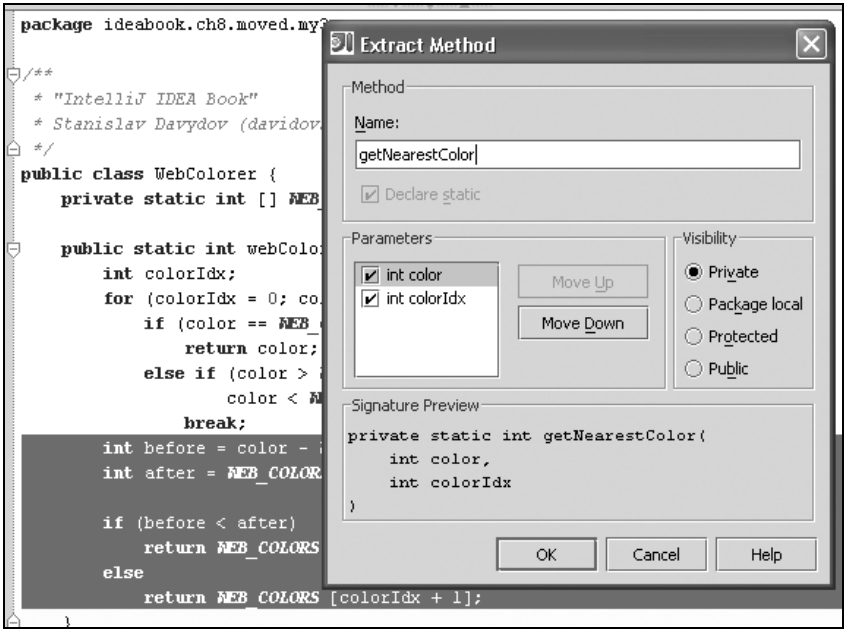


Рис. 8.49. Рефакторинг выделения метода

В открывшемся диалоге имеется возможность указать название создаваемого метода (**Name**), его параметры (**Parameters**), а также область видимости (**Visibility**).

После выполнения рефакторинга код класса будет преобразован к виду, представленному в листинге 8.9.

Листинг 8.9

```
public class WebColorer {
    private static int [] WEB_COLORS = {
        0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF};

    public static int webColor (int color) {
        int colorIdx;
        for (colorIdx = 0; colorIdx < WEB_COLORS.length; colorIdx++)
            if (color == WEB_COLORS [colorIdx])
                return color;
            else if (color > WEB_COLORS[colorIdx] &&
                    color < WEB_COLORS [colorIdx+1])
                break;
        return getNearestColor(color, colorIdx);
    }

    private static int getNearestColor(int color, int colorIdx) {
        int before = color - WEB_COLORS [colorIdx];
        int after = WEB_COLORS [colorIdx + 1] - color;

        if (before < after)
            return WEB_COLORS [colorIdx];
        else
            return WEB_COLORS [colorIdx + 1];
    }
}
```

Замечание

IntelliJ IDEA не всегда позволяет выделить в метод произвольный кусок кода. Ограничением может являться несколько точек выхода вроде операторов `break` и `return`.

8.9. Рефакторинг *Replace Method Code Duplicates (Заменить методом повторяющийся код)*

Данный рефакторинг позволяет заменить все вхождения повторяющегося кода вызовом метода.

Продолжим рассмотрение кода класса, представленного в листинге 8.9. Добавим в него еще один метод, представленный в листинге 8.10.

Листинг 8.10

```
private static int getWebColor (int idx) {
    return WEB_COLORS[idx];
}
```

Если вызвать данный рефакторинг, установив курсор на метод из листинга 8.8, IntelliJ IDEA найдет повторяющиеся места в коде, которые можно заменить вызовом данного метода (рис. 8.50).

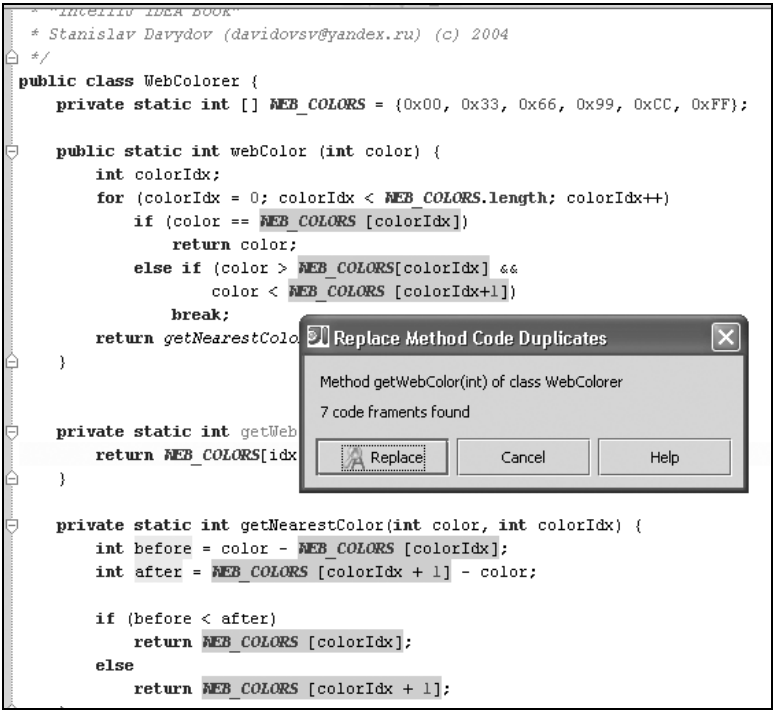


Рис. 8.50. Найдено несколько мест в коде, для которых можно произвести рефакторинг

Если подтвердить рефакторинг, нажав кнопку **Replace** (Заменить), код класса последовательно преобразуется к виду, представленному в листинге 8.11.

Листинг 8.11

```
public class WebColorer {
    private static int [] WEB_COLORS = {
        0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF};

    public static int webColor (int color) {
        int colorIdx;
        for (colorIdx = 0; colorIdx < WEB_COLORS.length; colorIdx++)
            if (color == getWebColor(colorIdx))
                return color;
            else if (color > getWebColor(colorIdx) &&
                    color < getWebColor(colorIdx+1))
                break;
        return getNearestColor(color, colorIdx);
    }

    private static int getWebColor (int idx) {
        return WEB_COLORS[idx];
    }

    private static int getNearestColor(int color, int colorIdx) {
        int before = color - getWebColor(colorIdx);
        int after = getWebColor(colorIdx + 1) - color;

        if (before < after)
            return getWebColor(colorIdx);
        else
            return getWebColor(colorIdx + 1);
    }
}
```

8.10. Рефакторинг *Introduce Variable (Ввести переменную)*

"Горячие" клавиши: <Ctrl>+<Alt>+<V>.

Рефакторинг позволяет преобразовать некоторое выражение в локальную переменную.

Рассмотрим работу данного рефакторинга на примере класса, представленного в листинге 8.12.

Листинг 8.12

```
public class Sum {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int c = a * b + a * b + a * b;
    }
}
```

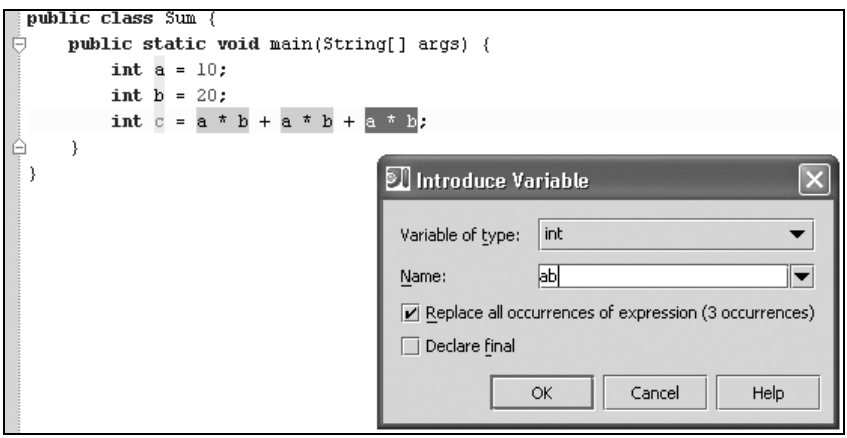


Рис. 8.51. Применение рефакторинга **Introduce Variable**

Выделим часть кода `a * b` и выполним команду рефакторинга (рис. 8.51).

В открывшемся диалоге можно указать:

- **Variable of type** (Переменная типа) — тип создаваемой переменной;
- **Name** (Имя) — имя переменной;
- **Replace all occurrences of expression** (Заменять все вхождения выражения).

При выполнении данного рефакторинга можно заменить только выделенное выражение. Но имеется и возможность заменить все повторяющиеся выражения. Для этого нужно установить опцию **Replace all occurrences of expression** (Заменять все вхождения выражения).

В результате выполнения рефакторинга код класса примет вид, представленный в листинге 8.13.

Листинг 8.13

```
public class Sum {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ab = a * b;
        int c = ab + ab + ab;
    }
}
```

8.11. Рефакторинг *Introduce Field (Ввести поле класса)*

"Горячие" клавиши: <Ctrl>+<Alt>+<F>.

Рефакторинг позволяет преобразовать некоторое выражение в поле класса.

8.11.1. Локальная переменная статического метода

Рассмотрим данный рефакторинг на примере класса из листинга 8.13. Подведем курсор к переменной `a` в третьей строчке и выполним команду рефакторинга (рис. 8.52).

В открывшемся диалоге можно задать следующие параметры преобразования:

- **Static field of type** (Тип статического поля класса);
- **Name** (Имя поля класса);
- **Initialize in** (Инициализировать в):
  - **Current method** (В текущем методе) — поле класса будет проинициализировано в текущем методе — там, где переменная присутствует в данный момент;
  - **Field declaration** (Определение поля) — поле класса будет инициализировано при определении. В случае выбора этой опции появится возможность добавить полю класса модификатор доступа `final` (**Declare final**);

- **Class constructor(s)** (Конструктор(ы) класса) — поле будет проинициализировано в конструкторе класса;

❒ **Visibility** (Видимость) — варианты видимости создаваемого поля класса: `private`, `package local`, `protected`, `public`;

❒ **Delete variable declaration** (Удалить определение переменной) — данная опция включена по умолчанию и не подлежит изменению.

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.14.

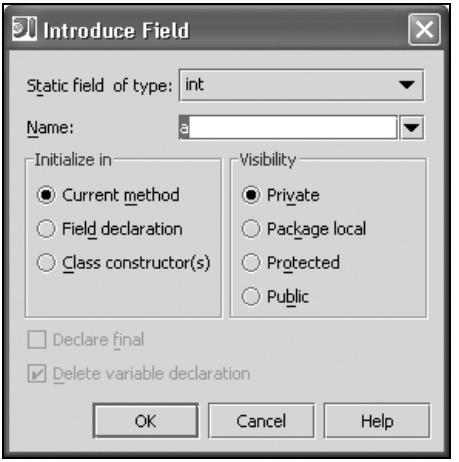


Рис. 8.52. Выполнение рефакторинга **Introduce Field** для переменной из статического метода

Листинг 8.14

```
public class Sum {
    private static int a;

    public static void main(String[] args) {
        a = 10;
        int b = 20;
        int ab = a * b;
        int c = ab + ab + ab;
    }
}
```

Локальная переменная, для которой мы проводили рефакторинг, определена в статическом методе `main()`. Поэтому поле класса также объявлено статическим.

8.11.2. Локальная переменная обычного метода

Рассмотрим класс, код которого представлен в листинге 8.15.

Листинг 8.15

```
public class Calculator {
    public double calculate () {
        double a = 123.0;
        double b = 215.0;
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

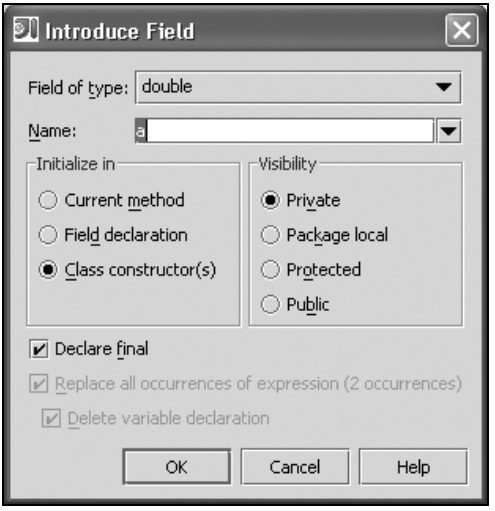


Рис. 8.53. Выполнение рефакторинга **Introduce Field** для переменной из обычного метода

Выполним рефакторинг для переменных `a` и `b` (рис. 8.53).

Набор опций в диалоге выполнения рефакторинга остался прежним за исключением дополнительной опции **Replace all occurrences of expression** (Заменить все вхождения выражения), включенной по умолчанию.

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.16.

Листинг 8.16

```
public class Calculator {
    private final double a;
    private final double b;

    public Calculator() {
        a = 123.0;
        b = 215.0;
    }

    public double calculate () {
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

8.12. Рефакторинг *Introduce Constant (Ввести константу)*

"Горячие" клавиши: <Ctrl>+<Alt>+<C>.

Рефакторинг позволяет преобразовать некоторое постоянное значение, встречающееся в коде, в именованную константу — поле класса, имеющее модификаторы `static final`.

Рассмотрим работу данного рефакторинга на примере кода из листинга 8.16.

Вызовем команду рефакторинга для значений `123.0` и `215.0` (рис. 8.54).

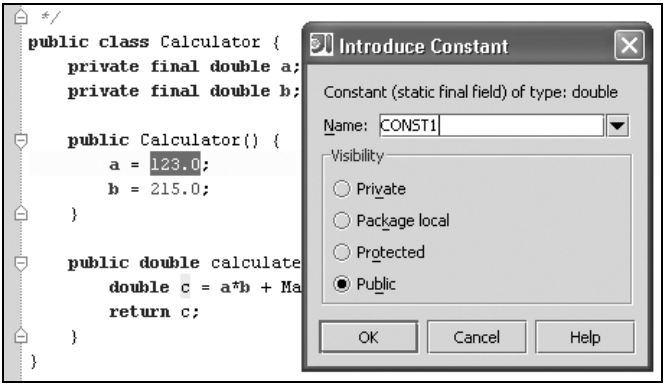


Рис. 8.54. Преобразование выражения в константу

В открывшемся диалоге можно ввести следующие параметры:

- **Name** (Имя) — имя константы;
- **Visibility** (Область видимости).

В результате преобразования код класса примет вид, представленный в листинге 8.17.

Листинг 8.17

```
public class Calculator {
    private final double a;
    private final double b;
    public static final double CONST1 = 123.0;
    public static final double CONST2 = 215.0;

    public Calculator() {
        a = CONST1;
        b = CONST2;
    }

    public double calculate () {
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

8.13. Рефакторинг *Introduce Parameter (Ввести параметр)*

"Горячие" клавиши: <Ctrl>+<Alt>+<P>.

Рефакторинг заменяет вхождения некоторого выражения на новый параметр метода.

Рассмотрим работу рефакторинга на примере класса из листинга 8.17. Применим рефакторинг для выражений `CONST1` и `CONST2` в теле конструктора класса (рис. 8.55).

Мы хотим добавить конструктору класса два новых параметра, которыми будут инициализироваться поля `a` и `b`. В результате выполнения рефакторинга код класса примет вид, представленный в листинге 8.18.

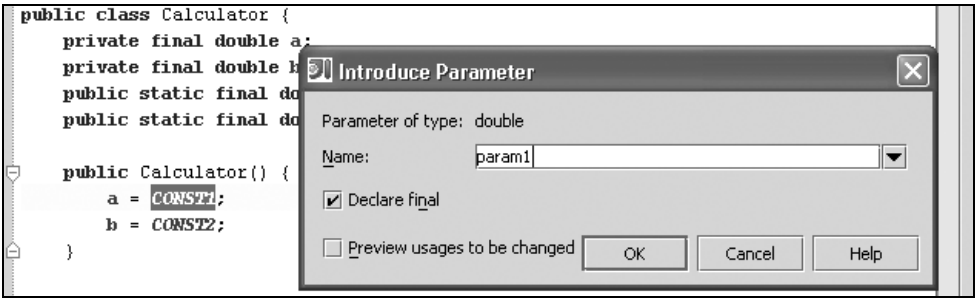


Рис. 8.55. Выделение параметра метода

Листинг 8.18

```
public class Calculator {
    private final double a;
    private final double b;
    public static final double CONST1 = 123.0;
    public static final double CONST2 = 215.0;

    public Calculator(final double param1, final double param2) {
        a = param1;
        b = param2;
    }

    public double calculate () {
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

8.14. Рефакторинг *Extract Interface (Выделить интерфейс)*

Рефакторинг выделения интерфейса из класса.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.18.

При выполнении команды рефакторинга IntelliJ IDEA отобразит диалог настроек преобразования (рис. 8.56).

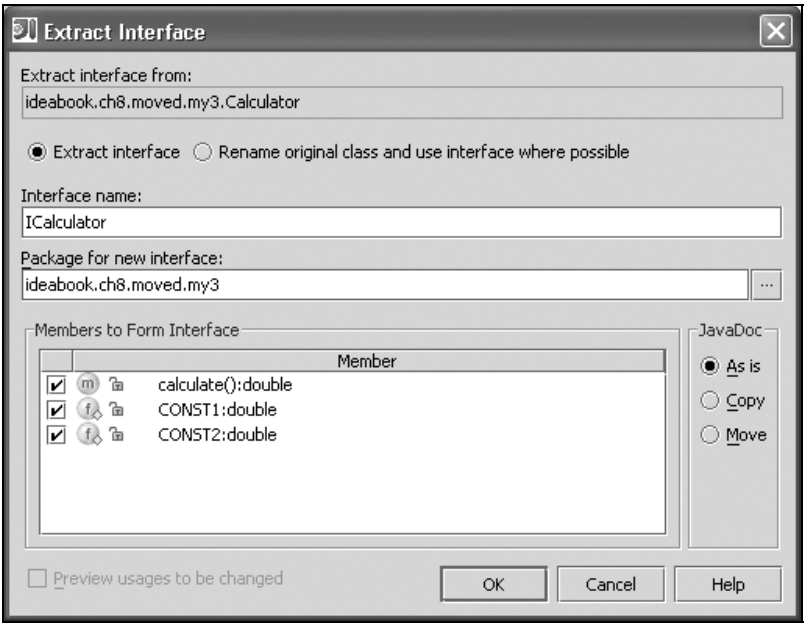


Рис. 8.56. Выделение интерфейса из методов класса

В данном диалоге можно настроить следующие опции и параметры:

- **Extract interface** (Выделить интерфейс) — в результате рефакторинга из методов класса будет выделен интерфейс с заданным названием;
- **Rename original class and use interface where possible** (Переименовать оригинальный класс и использовать интерфейс везде, где только возможно) — в результате рефакторинга оригинальный класс будет переименован, кроме того, будет создан интерфейс с именем метода, а все использования класса будут заменены использованием интерфейса везде, где только это возможно;
- **Interface name/Rename original class to** (Имя интерфейса/Новое имя оригинального класса при переименовании);
- **Package for new interface** (Пакет для нового интерфейса);
- **Members to Form Interface** (Члены класса для формирования интерфейса) — в данном списке отображены все методы и поля класса, доступные для вынесения в интерфейс;
- **JavaDoc** — комментарии JavaDoc в интерфейсе:
  - **As is** (Оставить как есть);
  - **Copy** (Копировать комментарии);
  - **Move** (Переместить комментарии);

- **Preview usages to be changed** (Предварительный просмотр изменений) — данная опция доступна в случае переименования оригинального класса.

Если в текущем модуле имеется несколько директорий с исходными файлами, при выполнении рефакторинга будет отображен диалог выбора конкретной директории (рис. 8.57).

После того как интерфейс создан, IntelliJ IDEA выведет диалог с запросом на изменения всех вхождений оригинального класса интерфейсом (рис. 8.58).

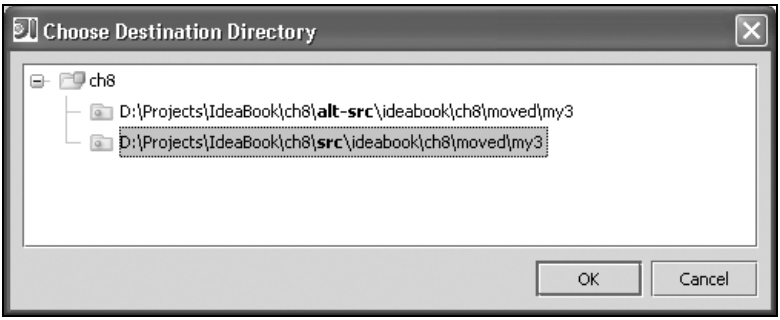


Рис. 8.57. Выбор директории с исходными файлами для создания интерфейса

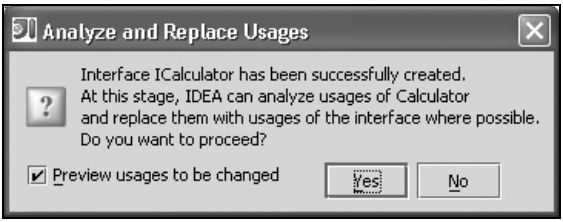


Рис. 8.58. Запрос на замену использований класса новым интерфейсом

В результате рефакторинга будет создан интерфейс, код класса тоже несколько изменится (листинг 8.19).

Листинг 8.19

```
public interface ICalculator {
    double CONST1 = 123.0;
    double CONST2 = 215.0;

    double calculate ();
}

public class Calculator implements ICalculator {
```

```
    private final double a;
    private final double b;

    public Calculator(final double param1, final double param2) {
        a = param1;
        b = param2;
    }

    public double calculate () {
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

8.15. Рефакторинг *Extract Superclass (Выделить базовый класс)*

Рефакторинг выделяет базовый класс из полей и методов класса.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.19.

При выполнении команды рефакторинга IntelliJ IDEA отобразит диалог с настройками параметров выделения базового класса (рис. 8.59).

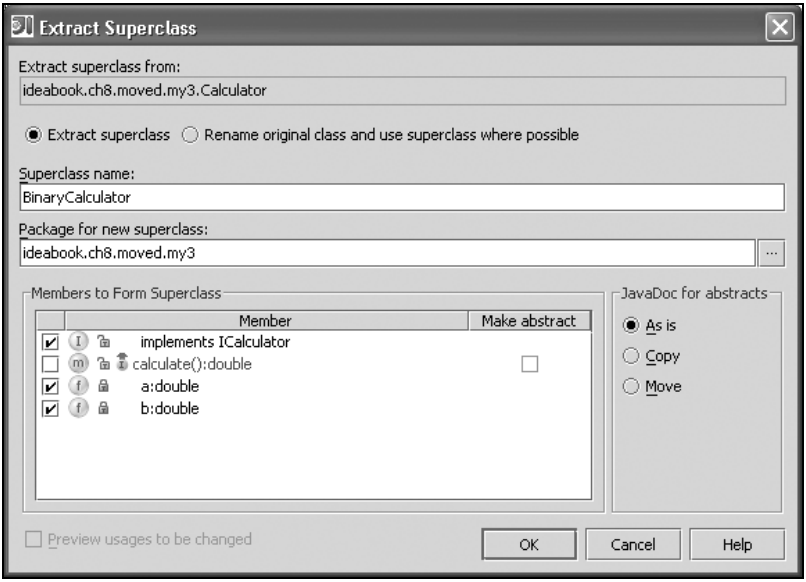


Рис. 8.59. Выделение базового класса

Работа данного рефакторинга похожа на работу рефакторинга **Extract Interface** (Выделить интерфейс) *(см. раздел 8.14)*.

С помощью данного диалога можно настроить следующие опции и параметры преобразования:

- **Extract superclass** (Выделить базовый класс) — в результате рефакторинга будет создан базовый класс из методов и полей данного класса;
- **Rename original class and use superclass where possible** (Переименовать оригинальный класс и использовать базовый класс везде, где это возможно) — в результате рефакторинга оригинальный класс будет переименован, и все вхождения оригинального класса будут заменены выделенным базовым;
- **Superclass name/Rename original class to** (Имя базового класса/Новое имя оригинального класса при переименовании);
- **Members to Form Superclass** (Члены класса для формирования базового класса);
- **JavaDoc for abstracts** (Комментарии в базовом классе):
  - **As is** (Не переносить);
  - **Copy** (Копировать из оригинального класса);
  - **Move** (Переместить из оригинального класса).

Если в текущем модуле имеется несколько директорий с исходными файлами, при выполнении рефакторинга будет отображен диалог выбора конкретной директории (см. рис. 8.57).

После того как базовый класс создан, IntelliJ IDEA выведет диалог с запросом на изменения всех вхождений оригинального класса базовым (рис. 8.60).

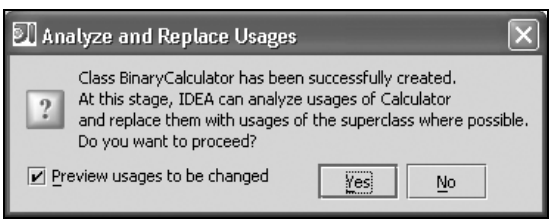


Рис. 8.60. Запрос на замену использований класса базовым классом

В результате рефакторинга будет создан базовый класс `BinaryCalculator`, а содержимое оригинального класса модифицировано (листинг 8.20).

Листинг 8.20

```
public abstract class BinaryCalculator implements ICalculator {
    protected final double a;
    protected final double b;
```

```
}

public class Calculator extends BinaryCalculator {
    public Calculator(final double param1, final double param2) {
        a = param1;
        b = param2;
    }

    public double calculate () {
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

Можно заметить, что конструктор класса `Calculator` стал содержать код, который не сможет скомпилироваться — присваивание значений полям базового класса, имеющим модификатор `final`, запрещено.

Придется немного модифицировать код, чтобы все было правильно (листинг 8.21).

Листинг 8.21

```
public abstract class BinaryCalculator implements ICalculator {
    protected final double a;
    protected final double b;

    protected BinaryCalculator(double a, double b) {
        this.a = a;
        this.b = b;
    }
}

public class Calculator extends BinaryCalculator {
    public Calculator(final double param1, final double param2) {
        super (param1, param2);
    }

    public double calculate () {
```

```
        double c = a*b + Math.sqrt(a*b);
        return c;
    }
}
```

8.16. Рефакторинг *Use Interface Where Possible (Использовать интерфейс, где это возможно)*

Рефакторинг заменяет места в коде, в которых используется класс, на один из его базовых классов или интерфейсов, если это возможно.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.22.

Листинг 8.22

```
public class CalculateSome {
    public static void main(String[] args) {
        Calculator c1 = new Calculator(10, 20);
        Calculator c2 = new Calculator(30, 10);

        double c = c1.calculate() + c2.calculate();
    }
}
```

Данный класс использует два объекта типа `Calculator`. Вызовем команду рефакторинга для класса `Calculator` из листинга 8.21 (рис. 8.61).

В представленном диалоге имеется возможность выбрать следующие опции:

- **Change usages of "..." to** (Заменять использования класса на один вариант из списка);
- **Use interface/superclass in instanceof** (Использовать интерфейс или базовый класс в операторах `instanceof`);
- **Preview usages to be changed** (Предварительный просмотр изменений).

В результате рефакторинга IntelliJ IDEA найдет два потенциальных места в коде, в которых можно заменить класс интерфейсом (рис. 8.62).

Выполнив рефакторинг, мы получим измененный класс `CalculateSome` (листинг 8.23).

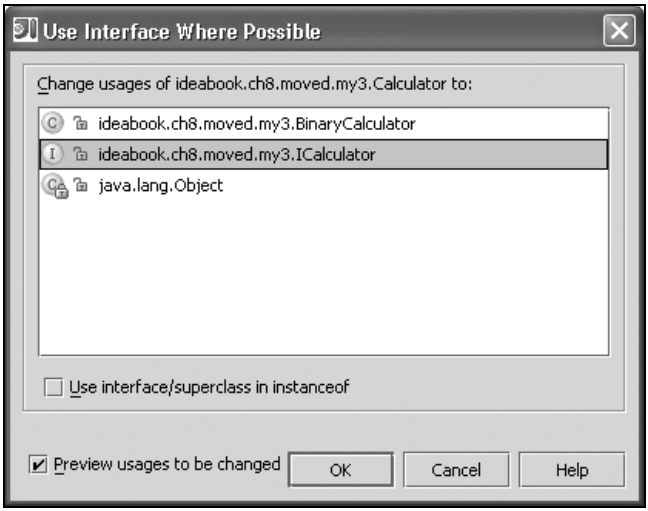


Рис. 8.61. Замена использований класса интерфейсом

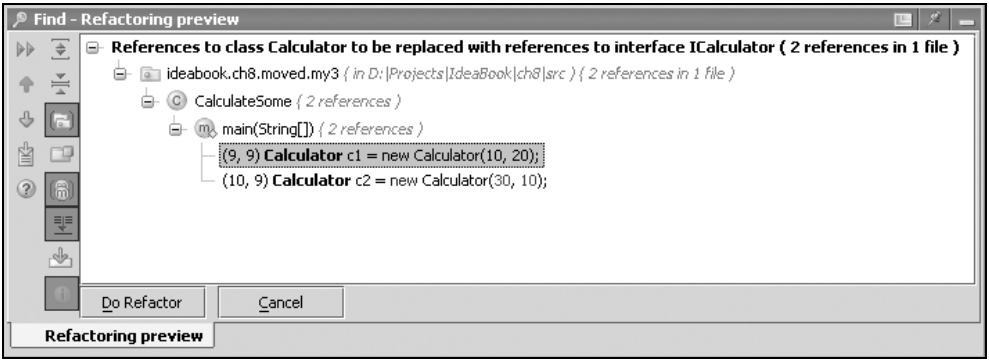


Рис. 8.62. Места в коде, в которых использование класса может быть заменено интерфейсом

Листинг 8.23

```
public class CalculateSome {
    public static void main(String[] args) {
        ICalculator c1 = new Calculator(10, 20);
        ICalculator c2 = new Calculator(30, 10);

        double c = c1.calculate() + c2.calculate();
    }
}
```

8.17. Рефакторинг *Pull Members Up (Перемещение членов класса вверх)*

Рефакторинг позволяет переместить члены класса (методы, поля) в его базовый класс.

Рассмотрим работу данного рефакторинга на двух классах из листинга 8.24.

Листинг 8.24

```
public class PlainFigure {
}

public class Square extends PlainFigure {
    private int length;
    private int width;
    private int perimeter;

    public Square(int length, int width) {
        this.length = length;
        this.width = width;
        setPerimeter(length * width);
    }

    protected void setPerimeter (int p) {
        perimeter = p;
    }
    public int getPerimeter() {
        return perimeter;
    }
}
```

Выполним рефакторинг для класса `Square` (рис. 8.63).

В открывшемся диалоге можно указать следующие опции:

- **Pull up members of "..." to** (Переместить члены выбранного класса в) — список доступных базовых классов для перемещения членов выбранного класса;
- **Members to be pulled up** (Члены класса для перемещения в базовый класс) — для методов класса имеется возможность переносить их в базо-

вый класс как абстрактные методы, для чего нужно установить флажок напротив соответствующего метода;

- **JavaDoc for abstracts** (Комментарии в базовом классе):
  - **As is** (Не переносить);
  - **Copy** (Копировать из оригинального класса);
  - **Move** (Переместить из оригинального класса).

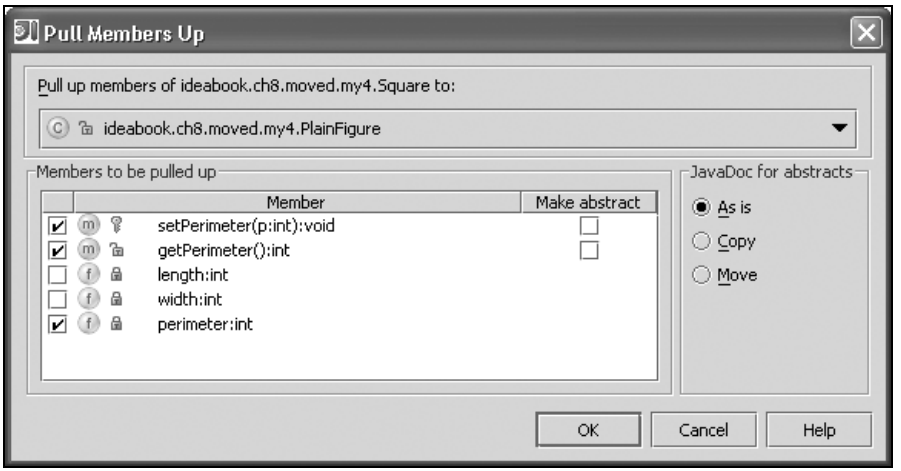


Рис. 8.63. Рефакторинг **Pull Members Up**

После выполнения рефакторинга код классов примет вид, представленный в листинге 8.25.

Листинг 8.25

```
public class PlainFigure {
    private int perimeter;

    protected void setPerimeter (int p) {
        perimeter = p;
    }
    public int getPerimeter() {
        return perimeter;
    }
}

public class Square extends PlainFigure {
```

```
    private int length;
    private int width;

    public Square(int length, int width) {
        this.length = length;
        this.width = width;
        setPerimeter(length * width);
    }
}
```

8.18. Рефакторинг *Push Members Down (Перемещение членов класса вниз)*

Рефакторинг переносит члены базового класса в один из его производных классов.

Данный рефакторинг, по сути, является обратным к рассмотренному в *разделе 8.17*. Рассмотрим работу данного рефакторинга на примере классов из листинга 8.25.

Кроме производного класса `Square` из листинга 8.25, мы создали класс `Circle`, который тоже наследуется от базового класса `PlainFigure` (листинг 8.26).

Листинг 8.26

```
public class Circle extends PlainFigure {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
        setPerimeter((int) (2 * Math.PI * radius));
    }
}
```

Применим рефакторинг к классу `PlainFigure` (рис. 8.64).

В диалоге настроек параметров рефакторинга можно указать поля и методы класса, которые необходимо переместить из базового класса. Также имеется возможность оставить методы в базовом классе, но сделать их абстрактными.

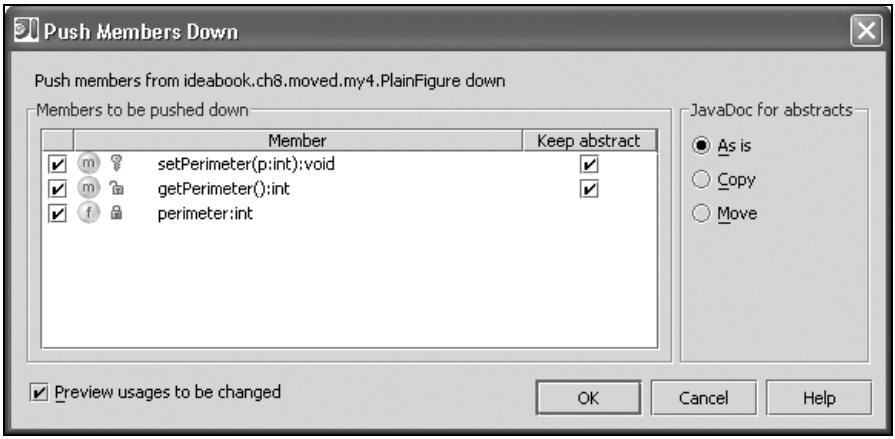


Рис. 8.64. Рефакторинг **Push Members Down**

В правой части диалога имеются опции, указывающие IntelliJ IDEA, как поступать с комментариями JavaDoc-членов базового класса (**JavaDoc for abstracts**):

- **As is** (Оставлять как есть);
- **Copy** (Копировать в класс, куда будут перемещены члены базового класса);
- **Move** (Переносить в класс, куда будут перемещены члены базового класса).

Также можно указать IntelliJ IDEA выводить предварительный список изменений (**Preview usages to be changed**), которые предполагает совершить рефакторинг (рис. 8.65).

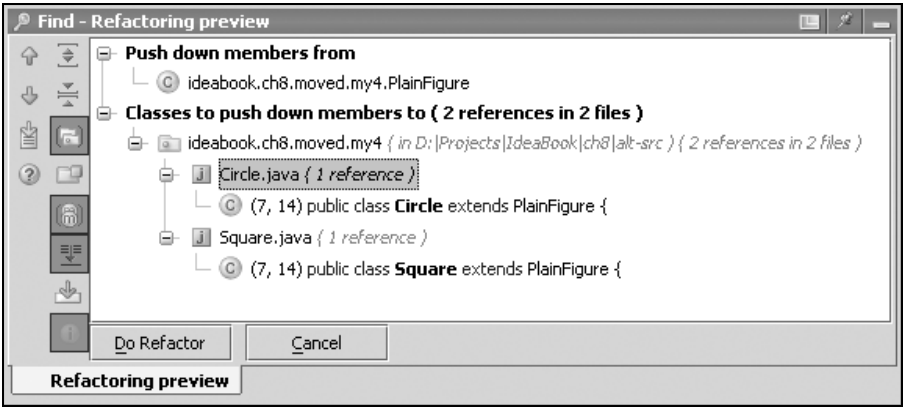


Рис. 8.65. Предварительный просмотр изменений рефакторинга

В результате выполнения рефакторинга код классов `PlainFugure`, `Square` и `Circle` примет вид, как в листинге 8.27.

Листинг 8.27

```
public abstract class PlainFigure {
    protected abstract void setPerimeter (int p);
    public abstract int getPerimeter();
}

public class Square extends PlainFigure {
    private int length;
    private int width;
    private int perimeter;

    public Square(int length, int width) {
        this.length = length;
        this.width = width;

        setPerimeter(length * width);
    }

    protected void setPerimeter (int p) {
        perimeter = p;
    }
    public int getPerimeter() {
        return perimeter;
    }
}

public class Circle extends PlainFigure {
    private int radius;
    private int perimeter;

    public Circle(int radius) {
        this.radius = radius;
        setPerimeter((int) (2 * Math.PI * radius));
    }

    protected void setPerimeter (int p) {
```

```
        perimeter = p;
    }

    public int getPerimeter() {
        return perimeter;
    }
}
```

Таким образом, работа рефакторинга привела к тому, что члены базового класса оказались перенесенными в каждый из производных классов.

8.19. Рефакторинг *Replace Inheritance With Delegation (Заменить наследование делегированием)*

Рефакторинг позволяет заменить наследование некоторых методов класса делегированием.

Рассмотрим работу данного рефакторинга на примере классов из листинга 8.25. Вызовем рефакторинг для класса `Square` (рис. 8.66).

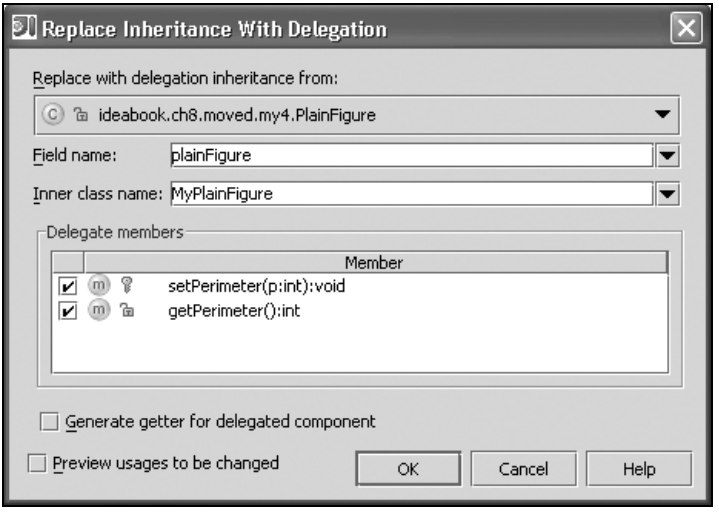


Рис. 8.66. Выбор методов для делегирования

В данном диалоге можно указать следующие опции для рефакторинга:

- **Field name** (Имя поля) — имя поля класса, которое будет использовано для вызова делегированных методов;

- **Inner class name** (Имя внутреннего класса) — имя внутреннего класса, с помощью которого будет выполнено делегирование;
- **Delegate members** (Делегировать члены класса) — список членов базового класса для делегирования;
- **Generate getter for delegated component** (Создать метод `get` для поля класса, через которое будут вызываться делегированные методы);
- **Preview usages to be changed** (Предварительный просмотр изменений).

В результате выполнения рефакторинга код класса `Square` примет вид, как в листинге 8.28.

Листинг 8.28

```
public class Square {
    private int length;
    private int width;
    private final MyPlainFigure plainFigure = new MyPlainFigure();

    public Square(int length, int width) {
        this.length = length;
        this.width = width;

        setPerimeter(length * width);
    }

    protected void setPerimeter(int p) {
        plainFigure.setPerimeter(p);
    }

    public int getPerimeter() {
        return plainFigure.getPerimeter();
    }

    private class MyPlainFigure extends PlainFigure { }
}
```

Данный рефакторинг удобно использовать тогда, когда появляется необходимость иметь наследование от двух классов и более. В этом случае наследование заменяется делегированием.

8.20. Рефакторинг *Inline (Подстановка в код)*

"Горячие" клавиши: <Ctrl>+<Alt>+<N>.

Рефакторинг заменяет вызов метода или использование статического поля класса на его тело или значение.

8.20.1. Подстановка для метода

Рассмотрим выполнение данного рефакторинга для метода на примере класса из листинга 8.11.

Применим данный рефакторинг к методу `getWebColor()` (рис. 8.67).

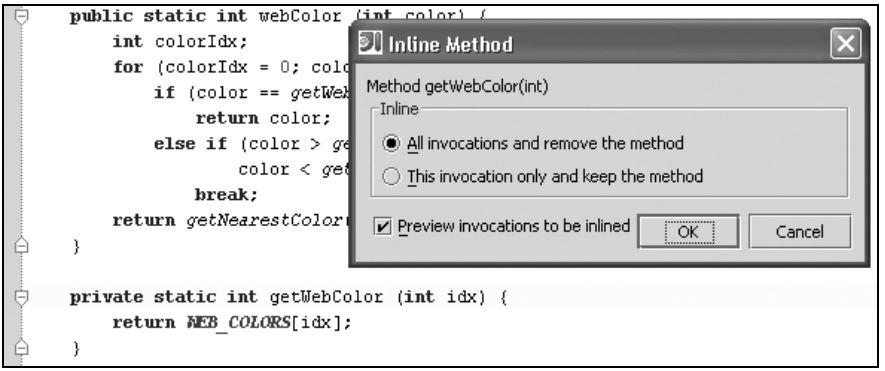


Рис. 8.67. Применение рефакторинга **Inline** к методу

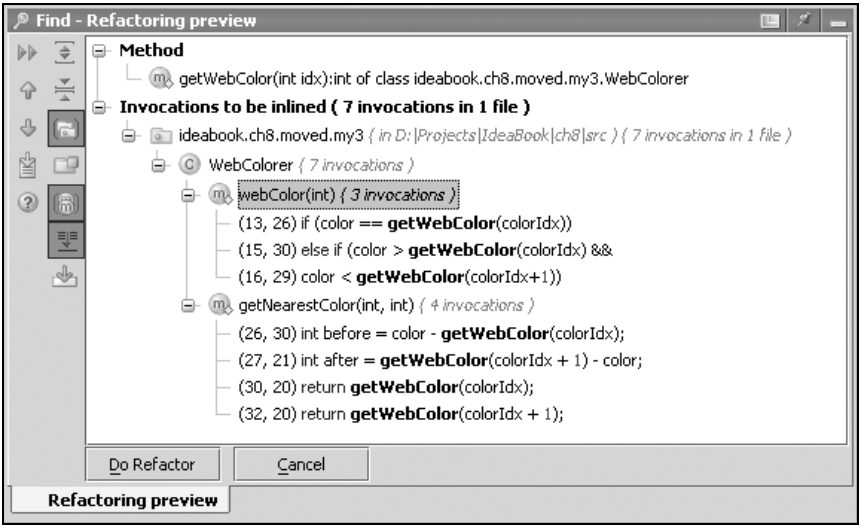


Рис. 8.68. Предварительный просмотр изменений

Имеется две возможности подстановки:

- **All invocations and remove the method** (Заменить все вызовы метода, а сам метод удалить);
- **This invocation only and keep the method** (Заменить только выбранный вызов, а все остальные оставить без изменения).

Также имеется возможность установить дополнительную опцию **Preview invocations to be inlined** (Предварительный просмотр вызовов, которые будут заменены). В нашем случае было найдено несколько вызовов метода, к которым будет применен рефакторинг (рис. 8.68).

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.29.

Листинг 8.29

```
public class WebColorer {
    private static final int [] WEB_COLORS = {
        0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF};

    public static int webColor (int color) {
        int colorIdx;
        for (colorIdx = 0; colorIdx < WEB_COLORS.length; colorIdx++)
            if (color == WEB_COLORS[colorIdx])
                return color;
            else if (color > WEB_COLORS[colorIdx] &&
                    color < WEB_COLORS[(colorIdx+1)])
                break;
        return getNearestColor(color, colorIdx);
    }

    private static int getNearestColor(int color, int colorIdx) {
        int before = color - WEB_COLORS[colorIdx];
        int after = WEB_COLORS[(colorIdx + 1)] - color;

        if (before < after) return WEB_COLORS[colorIdx];
        else return WEB_COLORS[(colorIdx + 1)];
    }
}
```

8.20.2. Подстановка для статического поля класса

Применение рефакторинга к статическому полю класса рассмотрим на примере класса из листинга 8.30.

Листинг 8.30

```
public class Circle extends PlainFigure {
    public static final double PI = 3.14;
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
        setPerimeter((int) (2 * PI * radius));
    }
}
```

Применим данный рефакторинг к статическому полю `PI` класса `Circle` (рис. 8.69).

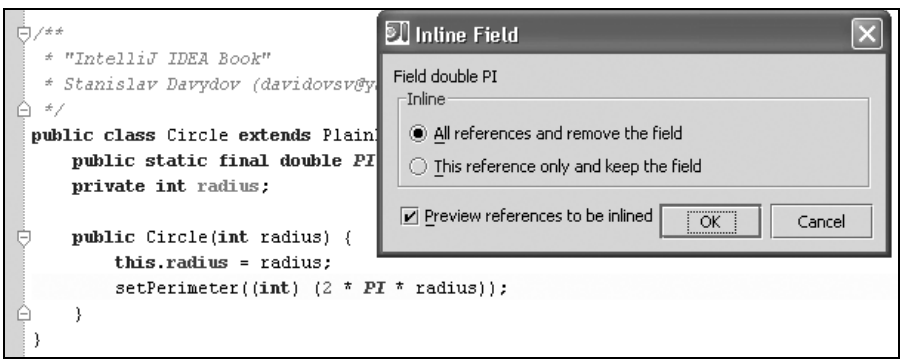


Рис. 8.69. Применение рефакторинга **Inline** к полю класса

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.31.

Листинг 8.31

```
public class Circle extends PlainFigure {
    private int radius;

    public Circle(int radius) {
```

```
        this.radius = radius;
        setPerimeter((int) (2 * 3.14 * radius));
    }
}
```

8.21. Рефакторинг *Convert Anonymous to Inner (Конвертировать анонимный класс во внутренний)*

Рефакторинг позволяет конвертировать анонимный класс во внутренний класс.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.32.

Листинг 8.32

```
public class MyPanel extends JPanel {
    public MyPanel() {
        JButton button = new JButton("Do!");
        final String message = "Done!";

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(JOptionPane.getRootFrame(),
                        message);
            }
        });
        add (button);
    }
}
```

В данном классе имеется анонимный класс, который мы преобразуем во внутренний (рис. 8.70).

В представленном диалоге можно настроить следующие параметры:

- **Class name** (Имя класса) — имя создаваемого внутреннего класса;
- **Make class static** (Сделать внутренний класс статическим);
- **Constructor Parameters** (Параметры конструктора) — параметры конструктора внутреннего класса.

Рис. 8.70. Преобразование анонимного класса во внутренний

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.33.

Листинг 8.33

```
public class MyPanel extends JPanel {
    public MyPanel() {
        JButton button = new JButton("Do!");
        final String message = "Done!";

        button.addActionListener(new MyActionListener(message));
        add (button);
    }

    private class MyActionListener implements ActionListener {
        private final String message;

        public MyActionListener(String message) {
            this.message = message;
        }

        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(JOptionPane.getRootFrame(),
                    message);
        }
    }
}
```

8.22. Рефакторинг *Encapsulate Fields (Инкапсулировать поля)*

Рефакторинг позволяет заменить прямое использование полей класса на вызовы соответствующих методов получения и установки значения.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.34.

Листинг 8.34

```
public class MyPanel2 extends JPanel {
    private JButton button;
    private JTextField textField;

    public MyPanel2() {
        button = new JButton("OK");
        textField = new JTextField();

        setLayout(new BorderLayout ());
        add(textField, BorderLayout.NORTH);
        add(button, BorderLayout.SOUTH);

        textField.setText("Text");
    }
}
```

В результате выполнения команды рефакторинга будет открыт диалог с параметрами (рис. 8.71).

В данном диалоге можно настроить следующие параметры:

- **Fields to Encapsulate** (Поля, которые нужно инкапсулировать);
- **Encapsulate** (Инкапсулировать):
  - **Get access** (Доступ на чтение);
  - **Set access** (Доступ на запись);
- **Use accessors even when field is accessible** (Использовать методы доступа даже тогда, когда поле доступно по области видимости);
- **Encapsulated Fields' Visibility** (Область видимости инкапсулируемых полей):
  - **Private**;

- **Package local**;
- **Protected**;
- **As is** (без изменения);

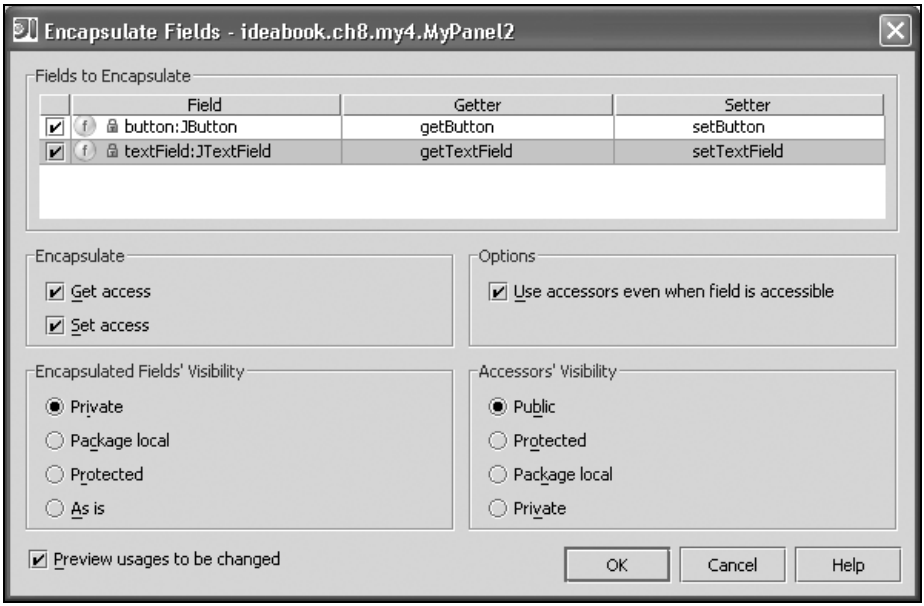


Рис. 8.71. Параметры рефакторинга **Encapsulate Fields**

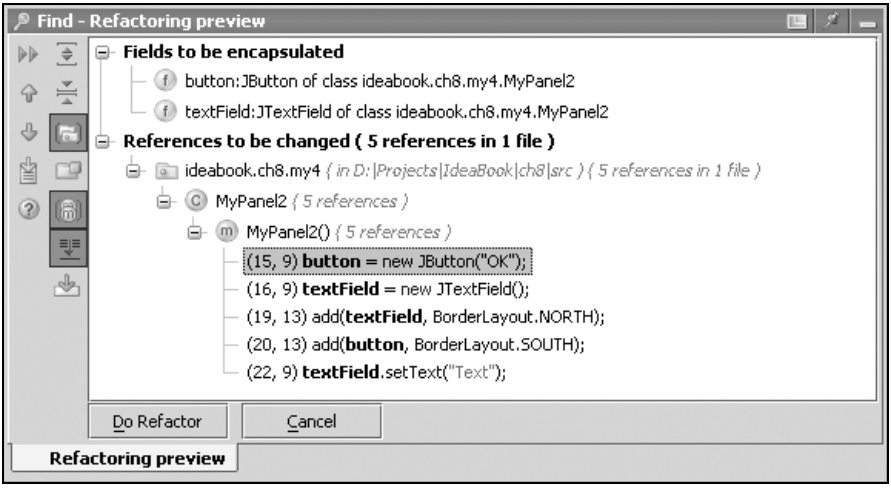


Рис. 8.72. Предварительный просмотр изменений

- **Accessors' Visibility** (Область видимости методов доступа):
  - **Public**;
  - **Protected**;
  - **Package local**;
  - **Private**;
- **Preview usages to be changed** (Предварительный просмотр изменений).

Рефакторинг определил следующие места использования полей класса, которые должны будут заменены вызовами методов доступа (рис. 8.72).

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.35.

Листинг 8.35

```
public class MyPanel2 extends JPanel {
    private JButton button;
    private JTextField textField;

    public MyPanel2() {
        setButton(new JButton("OK"));
        setTextField(new JTextField());

        setLayout(new BorderLayout ());
        add(getTextField(), BorderLayout.NORTH);
        add(getButton(), BorderLayout.SOUTH);

        getTextField().setText("Text");
    }

    public JButton getButton() {
        return button;
    }
    public void setButton(JButton button) {
        this.button = button;
    }
    public JTextField getTextField() {
        return textField;
    }
```

```
    public void setTextField(JTextField textField) {
        this.textField = textField;
    }
}
```

8.23. Рефакторинг *Replace Temp with Query (Заменить временное использование вызовом)*

Рефакторинг заменяет использование локальной переменной, значение которой не меняется после инициализации, на вызов метода, возвращающего ее значение.

Рассмотрим работу данного рефакторинга на примере класса, представленного в листинге 8.36.

Листинг 8.36

```
public class MsgPanel extends JPanel {
    private final JLabel message;

    public MsgPanel() {
        message = new JLabel ();
        add(message);
    }

    public void setMessage (String msg) {
        message.setText(msg);
    }

    public static final String [] MSGS = {
         "{0}: Successful", "{0}: Failed"};

    public void setMessageFor (String messagePart, int code) {
        String str = MessageFormat.format(MSGS[code],
                        new Object [] {messagePart});
        setMessage(str);
    }
}
```

Применим данный рефакторинг к локальной переменной `str` из метода `setMessageFor()`.

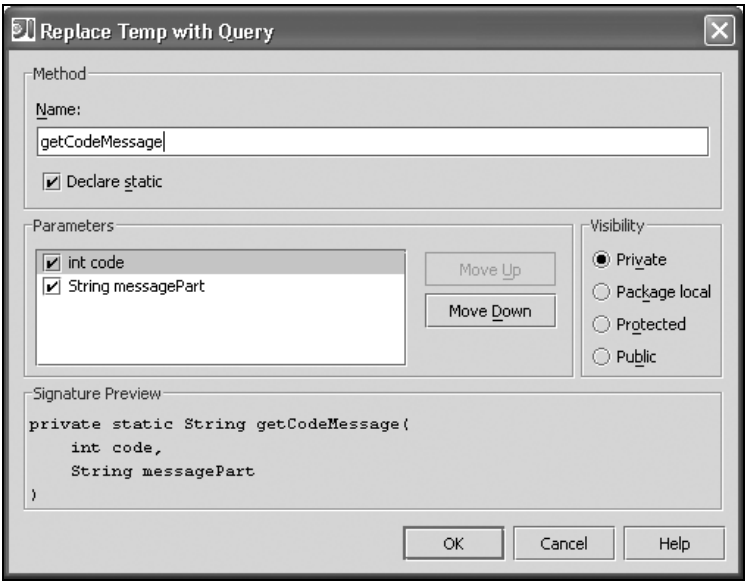


Рис. 8.73. Настройка параметров рефакторинга **Replace Temp with Query**

С помощью диалога, представленного на рис. 8.73, можно настроить дополнительные параметры рефакторинга:

- **Method** (Метод) — настройки для создаваемого метода:
  - **Name** (Имя метода);
  - **Declare static** (Объявить метод статическим);
- **Parameters** (Параметры метода);
- **Visibility** (Область видимости метода).

В нижней части диалога находится панель предварительного просмотра создаваемого метода.

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.37.

Листинг 8.37

```
public class MsgPanel extends JPanel {
    private final JLabel message;

    public MsgPanel() {
```

```
        message = new JLabel ();
        add(message);
    }

    public void setMessage (String msg) {
        message.setText(msg);
    }

    public static final String [] MSGS = {
         "{0}: Successful", "{0}: Failed"};

    public void setMessageFor (String messagePart, int code) {
        setMessage(getCodeMessage(code, messagePart));
    }

    private static String getCodeMessage(int code, String messagePart) {
        return MessageFormat.format(MSGS[code],
                new Object [] {messagePart});
    }
}
```

8.24. Рефакторинг *Replace Constructor With Factory Method (Заменить конструктор методом фабрики)*

Рефакторинг заменяет конструктор класса на метод фабрики. Конструктор при этом получает спецификацию доступа `private`, а в класс добавляется новый статический метод, вызывающий этот конструктор.

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.31.

Применим данный рефакторинг к конструктору этого класса (рис. 8.74).

В диалоге с параметрами рефакторинга можно задать:

- **Factory method name** (Имя метода фабрики);
- **In (fully qualified name)** — класс, в котором будет создан метод фабрики;
- **Preview usages to be changed** (Предварительный просмотр использований конструктора, которые будут изменены).

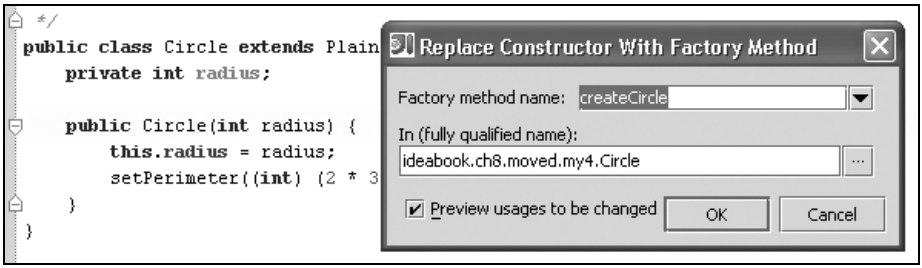


Рис. 8.74. Применение рефакторинга **Replace Constructor With Factory Method**

В результате применения рефакторинга код класса примет вид, представленный в листинге 8.38.

Листинг 8.38

```
public class Circle extends PlainFigure {
    private int radius;

    private Circle(int radius) {
        this.radius = radius;
        setPerimeter((int) (2 * 3.14 * radius));
    }

    public static Circle createCircle(int radius) {
        return new Circle(radius);
    }
}
```

8.25. Рефакторинг *Generify (Генерификация)*

Рефакторинг позволяет заменить старую нотацию языка Java на новую, использующую параметризованные классы (generics), появившиеся в пятой версии языка Java (Java 5). Подробнее о работе с параметризованными классами в IntelliJ IDEA можно прочитать в *главе 11.*

Данный рефакторинг доступен в случае, если модуль, которому принадлежит класс, поддерживает возможности языка Java 5, т. е. либо используется JDK 5.0, либо имеет установленную поддержку параметризованных классов (*см. раздел 3.2.11.6*).

Рассмотрим работу данного рефакторинга на примере класса из листинга 8.39.

Листинг 8.39

```
public class MyPanel3 extends JPanel {
    private List buttons;
    private ButtonGroup buttonGroup;

    public MyPanel3() {
        buttons = new ArrayList();
        buttonGroup = new ButtonGroup();
        placeButtons();
    }

    private void placeButtons () {
        removeAll();
        for (int i = 0; i < buttons.size(); i++) {
            JRadioButton button = (JRadioButton) buttons.get(i);
            add(button);
        }
    }

    public void addButtons (String [] names) {
        for (int i = 0; i < names.length; i++) {
            String name = names[i];
            JRadioButton button = new JRadioButton(name);
            buttonGroup.add(button);
            buttons.add(button);
        }
        placeButtons();
    }
}
```

При выполнении данного рефакторинга IntelliJ IDEA отобразит диалог с опциями, представленный на рис. 8.75.

В диалоге имеются следующие опции:

- **Drop obsolete casts** (Убирать ненужные приведения типов);
- **Leave Object-parameterized types raw** (Оставлять классы, одним из параметров которых является тип `java.lang.Object`, без изменения);

- **Preserve raw arrays** (Не заменять массивы параметризованными массивами). Данная опция имеется в диалоге настроек неслучайно. Дело в том, что компилятор, поставляемый в комплекте пакета предварительного доступа JSR-014 (*см. раздел 3.2.11.6*), не поддерживает данную возможность языка, в то время как компилятор в JDK 5.0 уже поддерживает (см. листинг 8.41);
- **Preview elements to be generified** (Предварительный просмотр изменений).

Рис. 8.75. Параметры рефакторинга **Generify**

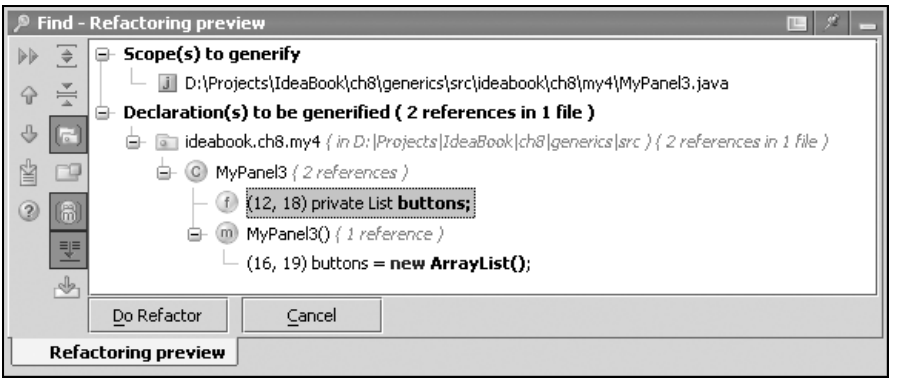


Рис. 8.76. Предварительный просмотр изменений рефакторинга

Рефакторинг определил изменения, представленные на рис. 8.76.

Измененный код класса представлен в листинге 8.40.

Листинг 8.40

```
public class MyPanel3 extends JPanel {
    private List<JRadioButton> buttons;
    private ButtonGroup buttonGroup;

    public MyPanel3() {
```

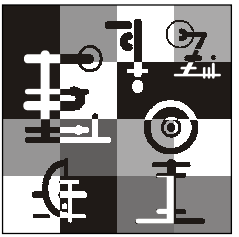
```
        buttons = new ArrayList<JRadioButton>();
        buttonGroup = new ButtonGroup();
        placeButtons();
    }

    private void placeButtons () {
        removeAll();
        for (int i = 0; i < buttons.size(); i++) {
            JRadioButton button = buttons.get(i);
            add(button);
        }
    }

    public void addButtons (String [] names) {
        for (int i = 0; i < names.length; i++) {
            String name = names[i];
            JRadioButton button = new JRadioButton(name);
            buttonGroup.add(button);
            buttons.add(button);
        }
        placeButtons();
    }

}
```

Данный рефакторинг может быть применен как к конкретным классам, так и к целым пакетам. При этом рефакторинг будет выполнен для всех классов из данного пакета, включая классы из вложенных пакетов.

В листинге 8.41 представлен код, на который рассчитана опция **Preserve array types** (Не изменять массивы параметризованными массивами).

Листинг 8.41

```
List<String>[] myStringArray;       // конструкция не поддерживается
                                    // компилятором из поставки JSR-14
                                    // см. раздел 3.2.11.6.
```



Глава 9

Компиляция, запуск, отладка

В данной главе будет рассмотрена компиляция классов, а также запуск и отладка различных типов приложений.

9.1. Компиляция

IntelliJ IDEA поддерживает два типа компиляторов:

1. Стандартный компилятор Javac фирмы Sun, поставляемый совместно с JDK.
2. Компилятор Jikes фирмы IBM.

По умолчанию используется компилятор Javac.

9.1.1. Компиляция Java-классов

Для компиляции Java-классов необходимо, чтобы модули и проект, в котором они находятся, имели настроенные параметры JDK (*см. разделы 5.2.1, 5.3.2*).

Для компиляции будет использован компилятор Javac из того JDK, который сконфигурирован для данного проекта или данного модуля.

Есть несколько вариантов запуска компиляции класса:

- компиляция класса как такового, при этом будет скомпилирован выбранный класс, а также классы, от которых он зависит;
- компиляция пакета, к которому относится класс, при этом кроме выбранного класса скомпилируются и остальные классы пакета, а также все классы, от которых зависят классы пакета;
- компиляция модуля, к которому относится класс, при этом будут скомпилированы все классы в модуле, а также классы из других модулей, от которых зависит данный модуль;
- компиляция всего проекта, при этом будут скомпилированы все классы.

IntelliJ IDEA умеет распознавать зависимости между классами на основе анализа кода, а также использует явные указания на зависимости между модулями, настроенными с помощью диалога настройки модулей проекта (*см. раздел 5.3.3*).

От того, какой командой запущена компиляция, зависит не только количество классов, которые будут скомпилированы. IntelliJ IDEA умеет распознавать классы, которые уже были скомпилированы и не обновлялись с момента последней компиляции. Если выполняется одна из разновидностей команды **Make** (Собрать) (*см. раздел 3.2.9*), такие классы не будут включены в компиляцию. Разумеется, классы, от которых зависят эти не обновлявшиеся классы, также должны быть не обновленными.

Если компиляция была вызвана командой **Rebuild** (Пересобрать) (*см. раздел 3.2.9.1*), будут заново скомпилированы все классы вне зависимости от того, обновлялись они или нет.

Если во время компиляции компилятор обнаруживает какие-либо ошибки или замечания, IntelliJ IDEA выводит информацию об этом в инструментальном окне **Messages** (Сообщения) (*см. раздел 3.4.8.1*) (рис. 9.1).

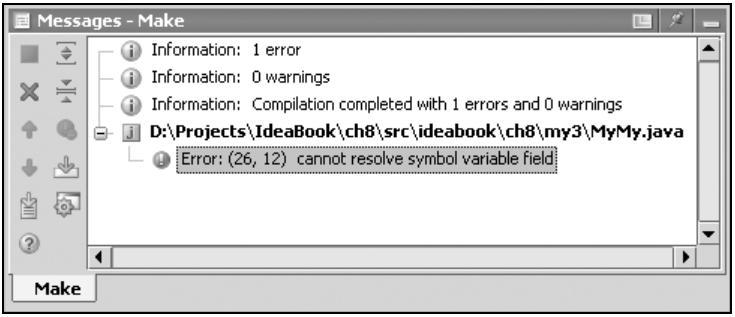


Рис. 9.1. Ошибки компиляции в инструментальном окне **Messages**

В IntelliJ IDEA имеется возможность выполнять компиляцию в фоновом режиме. Есть два способа выполнить такую фоновую компиляцию. Во-первых, во время компиляции можно нажать на кнопку **Background** (Фон) в диалоге с информацией о компиляции (рис. 9.2).

Во-вторых, можно включить соответствующую опцию в диалоге настройки компилятора (*см. раздел 4.2.2.1*).

Рис. 9.2. Диалог информации о ходе компиляции

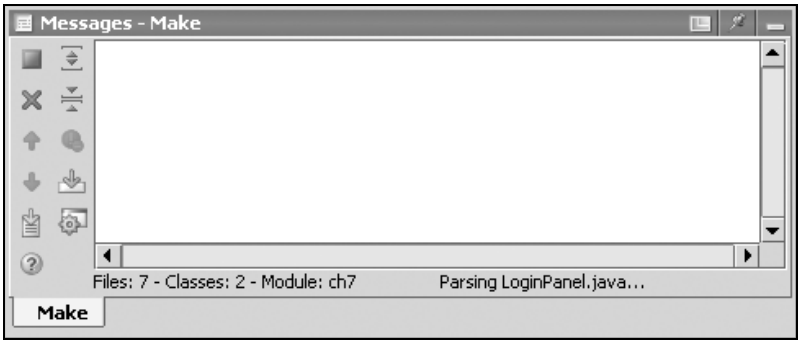


Рис. 9.3. Информация о компиляции в фоновом режиме

Фоновая компиляция хоть и изрядно поедает вычислительные ресурсы, однако позволяет не терять времени при компиляции больших проектов.

При фоновой компиляции информация о компиляции выводится в инструментальном окне **Messages** (Сообщения) (рис. 9.3). При этом можно прервать компиляцию с помощью кнопки **Stop** (Стоп) на панели инструментов окна сообщений.

9.1.2. Компиляция RMI

В IntelliJ IDEA имеется возможность автоматически выполнять компиляцию имплементаций RMI-интерфейсов, для чего в настройках компилятора имеется отдельная закладка свойств RMI-компилятора (*см. раздел 4.2.2.3*).

Для выполнения RMI-компиляции необходимо, чтобы была разрешена такая компиляция в настройках компилятора проекта. Если в проекте имеются классы, имплементирующие интерфейс `java.rmi.Remote`, для них будет автоматически запущен компилятор RMI.

9.1.3. Валидация JSP

В IntelliJ IDEA имеется возможность одновременно с запуском компиляции выполнять и валидацию JSP-файлов в Web-модулях, если таковые имеются и если для них включены соответствующие настройки (*см. раздел 5.3.6*).

9.2. Типы приложений

IntelliJ IDEA поддерживает запуск и отладку нескольких типов приложений:

- обычные Java-приложения, которые запускаются с помощью метода `main()`;

- апплеты — выполняются в контексте Web-браузера;
- тесты JUnit — выполняются с помощью утилиты JUnit;
- приложения, выполняющиеся на сервере приложений:
  - Tomcat;
  - WebLogic;
  - на произвольном сервере приложений, поддерживающем спецификацию JSR-45.

Также имеется возможность производить удаленную отладку любого поддерживающего такую отладку Java-приложения.

Для настройки параметров запуска и отладки используются специальные конфигурации, настраиваемые с помощью диалога настройки конфигураций запуска и отладки (рис. 9.4).

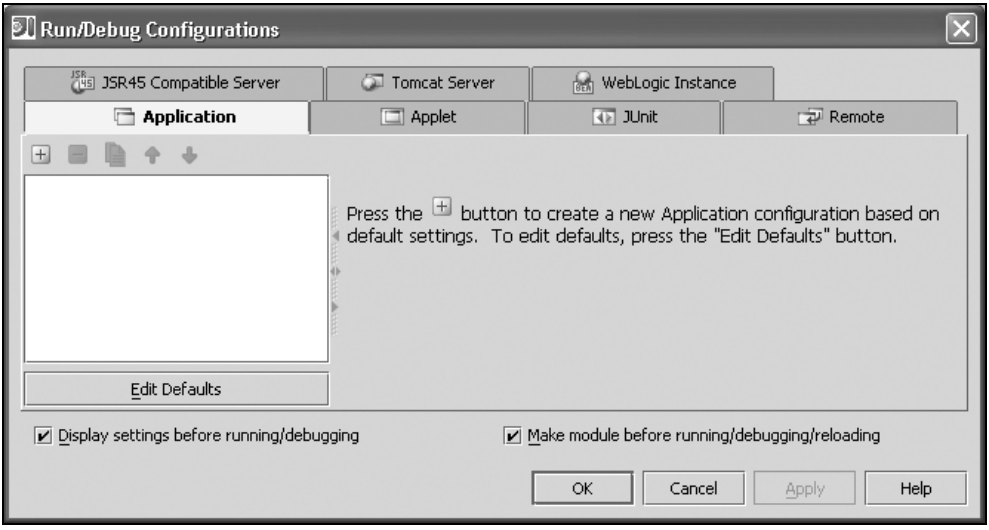


Рис. 9.4. Диалог настройки конфигураций запуска и отладки

Диалог настройки конфигурации можно открыть с помощью команды меню **Run | Edit Configurations** (Запуск | Редактирование конфигураций), а также с помощью команды из панели инструментов IntelliJ IDEA (*см. раздел 3.3*).

Данный диалог имеет общую структуру настройки для каждого типа конфигурации приложений. Основную часть диалога занимают закладки с настройками, соответствующие каждому из семи описанных типов. В левой верхней части закладки расположена панель инструментов, на которой

имеются кнопки для добавления, удаления, копирования конфигураций. В левой части закладки расположен список доступных конфигураций (на рис. 9.4 он пуст). Под списком расположена кнопка **Edit Defaults** (Редактировать настройки по умолчанию), которая открывает диалог с настройками по умолчанию.

В нижней части диалога располагаются две опции:

- **Display settings before running/debugging** (Показывать диалог настроек перед запуском или отладкой приложений);
- **Make module before running/debugging/reloading** (Собирать модуль перед запуском, отладкой или перезагрузкой).

Далее мы подробно рассмотрим все типы конфигураций.

9.2.1. Java-приложения (закладка *Application*)

На рис. 9.5 представлен общий вид диалога настройки конфигурации для Java-приложений.

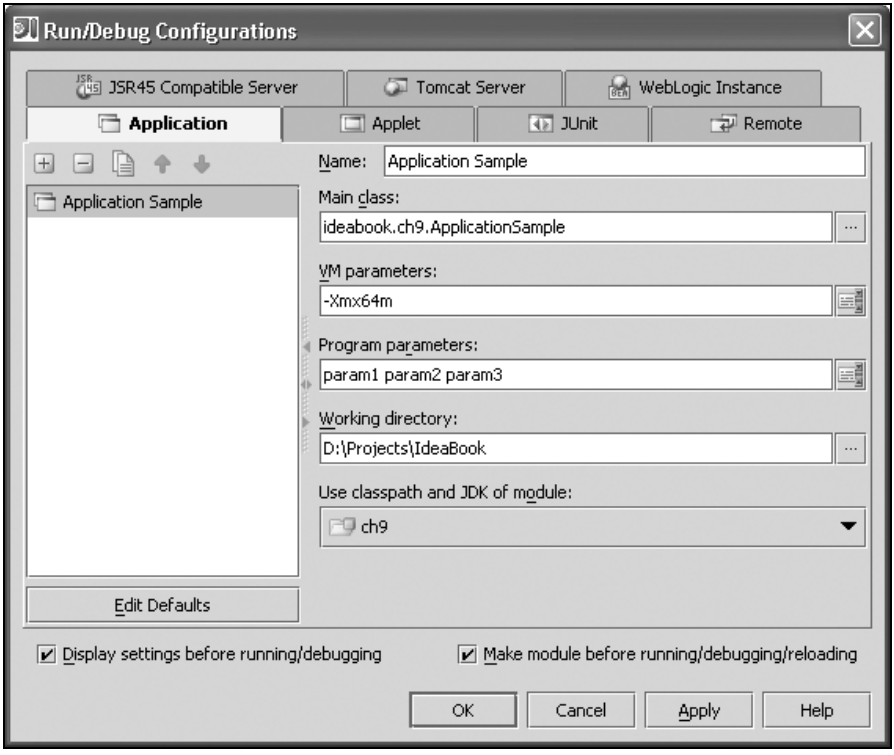


Рис. 9.5. Настройка конфигурации Java-приложений

В конфигурации Java-приложения имеется возможность задать следующие настройки:

- **Name** (Имя) — некоторое имя Java-приложения, с помощью которого можно осуществлять быстрый доступ к конфигурации для запуска и отладки;
- **VM parameters** (Параметры VM) — дополнительные параметры JVM, необходимые для запуска приложения;
- **Program parameters** (Параметры программы) — параметры для передачи в качестве аргументов в функцию `main()`;
- **Working directory** (Рабочая директория) — директория, в контексте которой будет запущено приложение;
- **Use classpath and JDK of module** (Использовать путь к классам и JDK модуля) — выпадающий список модулей, один из которых должен быть выбран для запуска приложения. Из настроек этого модуля будут взяты данные о классах для запуска приложения (classpath), а также о JDK. Приложение будет запущено под тем JDK, который настроен для выбранного модуля.

После того как конфигурация добавлена, приложение появится в списке доступных конфигураций для запуска и отладки (рис. 9.6).

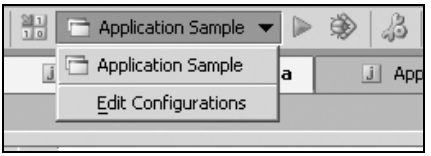


Рис. 9.6. Приложение в списке доступных конфигураций для запуска и отладки

9.2.2. Апплеты (закладка *Applet*)

На рис. 9.7 представлен общий вид диалога настройки конфигурации для Java-апплетов.

С помощью переключателей в верхней части закладки можно выбрать способ настройки конфигурации Java-апплета:

- **Applet class** (Класс апплета) — настройка по классу Java-апплета (рис. 9.7);
- **URL** — настройка по адресу HTML-файла, в котором содержится информация об апплете.

Рассмотрим оба варианта.

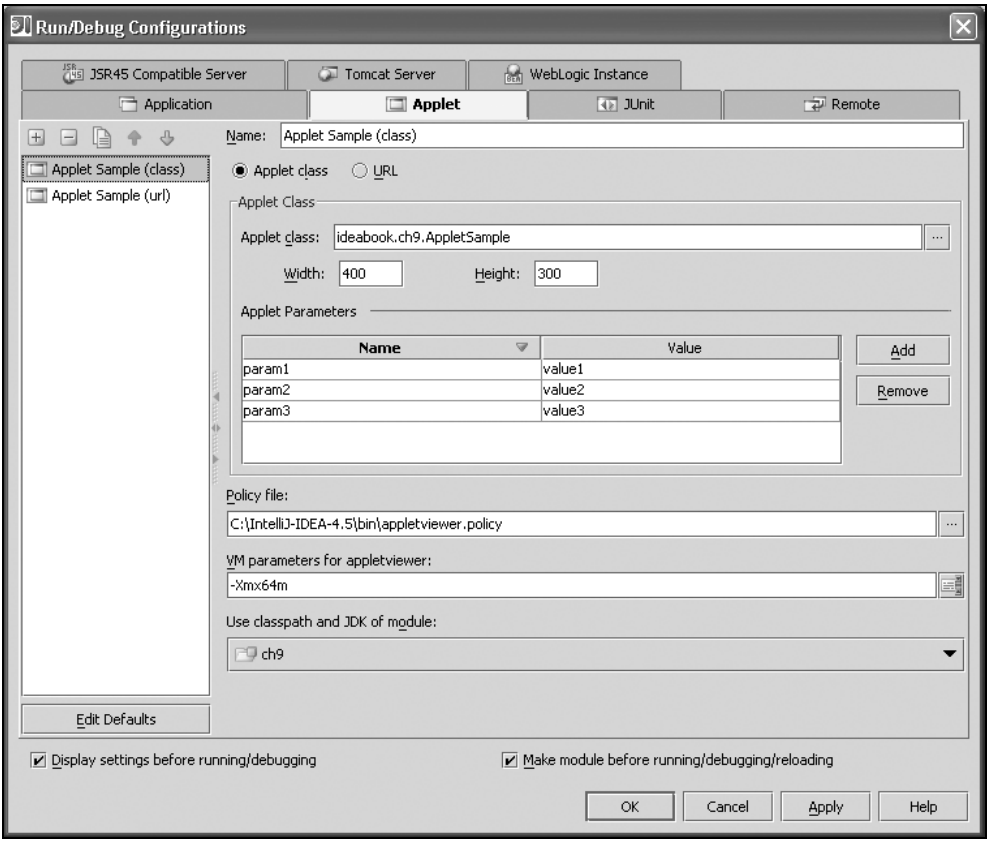


Рис. 9.7. Настройка конфигурации Java-апплета по классу

9.2.2.1. *Applet class (Класс апплета)*

При настройке конфигурации для класса апплета имеется возможность задать следующие опции и параметры:

- **Applet class** (Класс апплета) — можно задать вручную либо выбрать из списка (рис. 9.8);
- **Width** (Ширина) — ширина апплета в пикселах;
- **Height** (Высота) — высота апплета в пикселах;
- **Applet Parameters** (Параметры апплета) — таблица параметров, которые должны передаваться апплету при запуске. Параметры задаются с помощью пары **Name** (Имя), **Value** (Значение). Параметры можно добавлять (с помощью кнопки **Add**) и удалять (с помощью кнопки **Remove**).

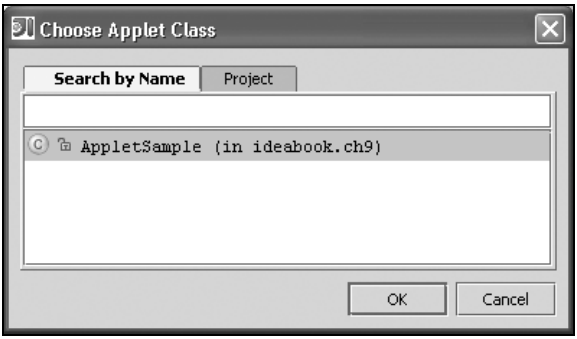


Рис. 9.8. Выбор класса апплета из списка

9.2.2.2. URL (Адрес апплета)

При настройке конфигурации для адреса апплета необходимо задать один-единственный параметр **URL/HTML file** (URL-адрес или HTML-файл) (рис. 9.9).

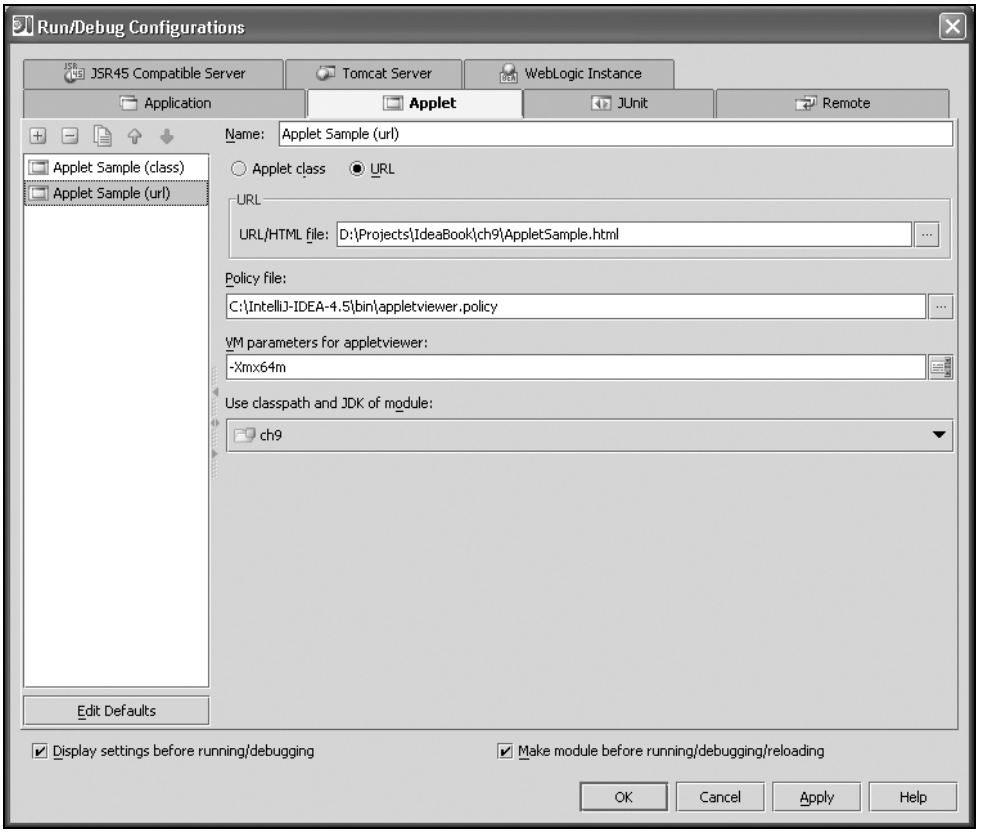


Рис. 9.9. Настройка конфигурации апплета по адресу

9.2.2.3. Общие параметры настройки апплета

Кроме параметров, описанных в *разделах 9.2.2.1—9.2.2.2*, имеются общие параметры для обоих типов настройки апплета:

- **Policy file** (Файл с правами доступа) — место расположения файла appletviewer.policy. Этот файл поставляется вместе с IntelliJ IDEA, поэтому данный параметр устанавливается по умолчанию. Однако, если требуется использовать для апплета иные права доступа, можно задать другой файл;
- **VM parameters for appletviewer** (Параметры VM для appletviewer);
- **Use classpath and JDK of module** (Использовать путь к классам и JDK модуля) — выпадающий список модулей, один из которых должен быть выбран для запуска апплета. Из настроек этого модуля будут взяты данные о классах для запуска апплета (classpath), а также о JDK. Апплет будет запущен под тем JDK, который настроен для выбранного модуля.

9.2.3. Тесты (закладка *JUnit*)

Общий вид диалога настройки конфигурации тестов представлен на рис. 9.10.

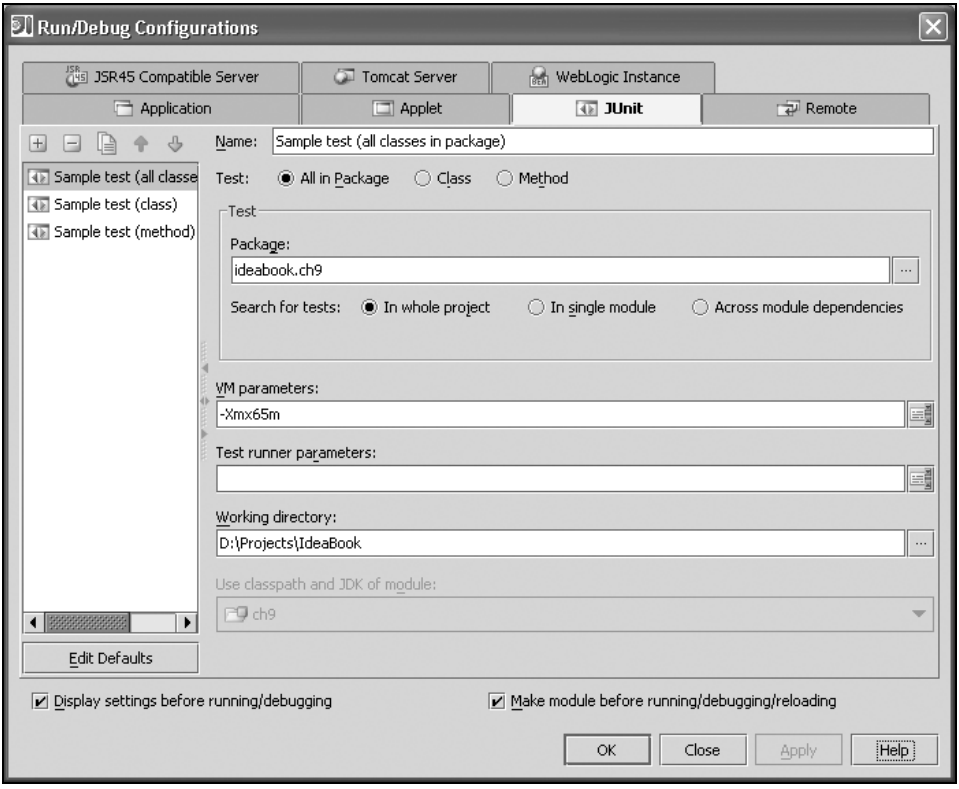


Рис. 9.10. Конфигурация тестов: все тесты в пакете

Для конфигурации тестов необходимо выбрать одну из трех возможностей:

- **All in Package** (Все тесты в пакете) — запуск всех тестов в указанном пакете;
- **Class** (Класс) — запуск тестов из указанного класса;
- **Method** (Метод) — запуск конкретного метода из класса-теста.

Каждый из приведенных вариантов конфигурации тестов имеет собственные настройки (*см. разделы 9.2.3.1—9.2.3.3*), также имеются и общие настройки (*см. раздел 9.2.3.4*).

9.2.3.1. Запуск всех тестов в пакете

Для запуска всех тестов в пакете необходимо настроить следующие параметры (см. рис. 9.10):

- **Package** (Название пакета). Название пакета можно задать вручную или выбрать из списка (рис. 9.11);
- **Search for tests** (Искать классы тестов в):
  - **In whole project** (Во всем проекте);
  - **In single module** (В отдельном модуле). При выборе данной опции необходимо обязательно задать модуль, в котором производить поиск (*см. раздел 9.2.3.4*);
  - **Across module dependencies** (По всем зависимостям модуля). При выборе данной опции также требуется задать модуль.

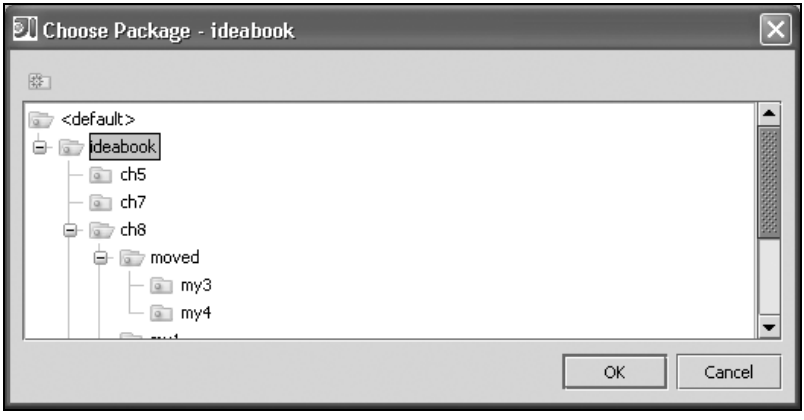


Рис. 9.11. Выбор пакета для запуска тестов

9.2.3.2. Запуск тестов одного класса

При запуске тестов из одного класса необходимо задать только один параметр **Class** (Класс) — класс, тесты которого нужно выполнять (рис. 9.12).

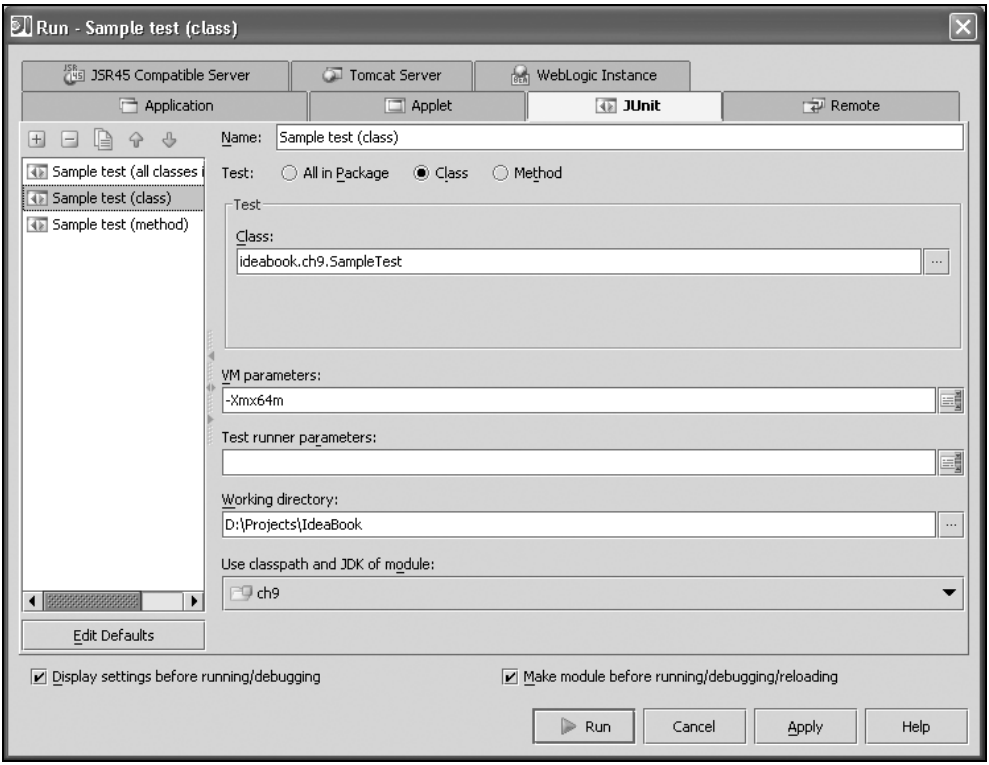


Рис. 9.12. Конфигурация тестов: все тесты класса

Для конфигурации данного типа также необходимо задать модуль (*см. раздел 9.2.3.4*).

9.2.3.3. Запуск конкретного метода из класса-теста

Для запуска конкретного метода из класса-теста необходимо указать два параметра (рис. 9.13):

- **Class** (Класс) — класс с тестами;
- **Method** (метод) — метод теста для запуска. Метод можно записать вручную, а можно выбрать из списка (рис. 9.14).

Для конфигурации теста данного типа также необходимо обязательно задать модуль (*см. раздел 9.2.3.4*).

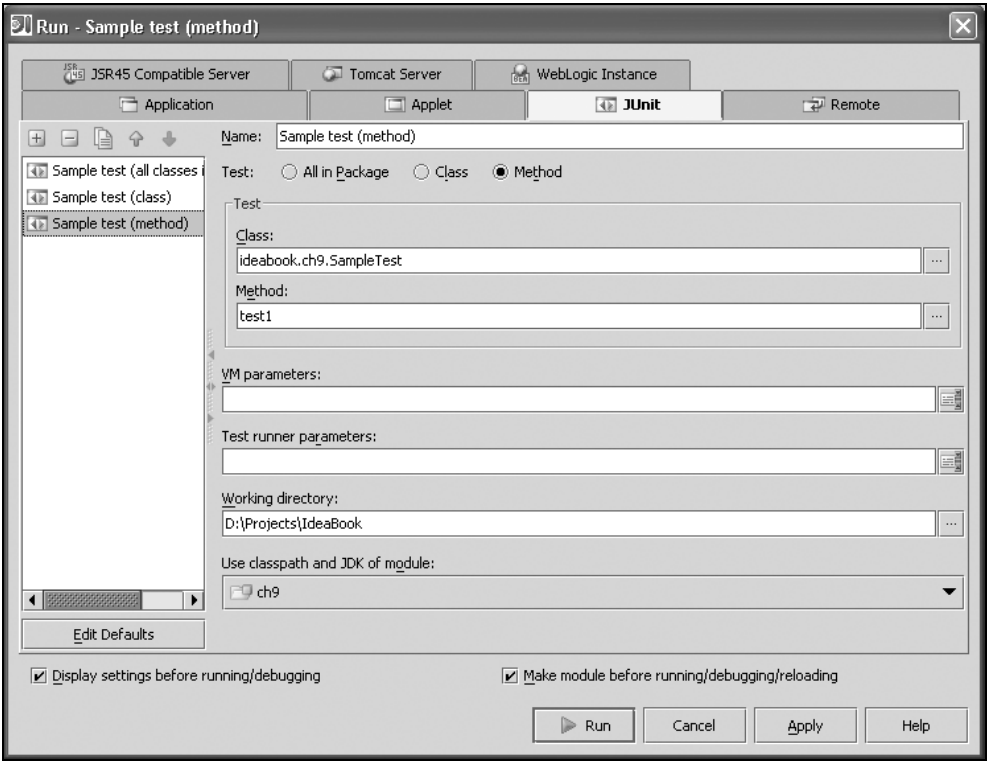


Рис. 9.13. Конфигурация тестов: конкретный метод класса-теста

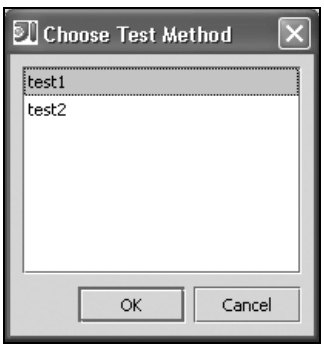


Рис. 9.14. Выбор метода с тестом из списка

9.2.3.4. Общие параметры настройки тестов

В конфигурации тестов также имеется возможность задать параметры, общие для каждого из вышеописанных типов тестов:

- **VM parameters** (Параметры VM) — дополнительные параметры виртуальной машины для запуска тестов;
- **Use classpath and JDK of module** (Использовать путь к классам и JDK модуля) — выпадающий список модулей, один из которых должен быть выбран для запуска тестов. Из настроек этого модуля будут взяты данные о классах для запуска апплета (classpath), а также о JDK. Тесты будут запущены под тем JDK, который настроен для выбранного модуля.

Настройка конкретного модуля не требуется при запуске всех тестов в пакете (**All in Package**), при выборке тестов из всего проекта (**Search for tests: In whole project**).

9.2.4. Удаленные приложения (закладка *Remote*)

Конфигурация удаленного приложения используется только для отладки. Общий вид конфигурации удаленного приложения представлен на рис. 9.15.

В верхней части панели расположено информационное поле, где указываются параметры, которые необходимо использовать для запуска удаленного приложения, чтобы оно поддерживало удаленную отладку.

Связь (**Transport**) с удаленным приложением возможна через соединения двух типов:

- **Socket** (Сокет) — связь по протоколу TCP/IP через сокет. В случае связи через сокет, необходимо задать два дополнительных параметра:
  - **Host** (Хост) — имя хоста, на котором запущено приложение;
  - **Port** (Порт) — номер порта, к которому будет подсоединяться отладчик;
- **Shared memory** (Разделяемая память) — связь через разделяемую память. Данный тип связи возможен только при отладке приложений, работающих под операционной системой Windows. Для соединения через разделяемую память необходимо указать дополнительный параметр **Shared memory address** (Адрес разделяемой памяти) (рис. 9.16).

В обоих режимах связи имеется возможность подключаться к отлаживаемому приложению (**Attach**) и запускать приложение в режиме прослушивания (**Listen**).

В режиме подключения (**Attach**) необходимо, чтобы отлаживаемое приложение уже было запущено. В режиме прослушивания (**Listen**) отладчик должен быть запущен перед запуском приложения.

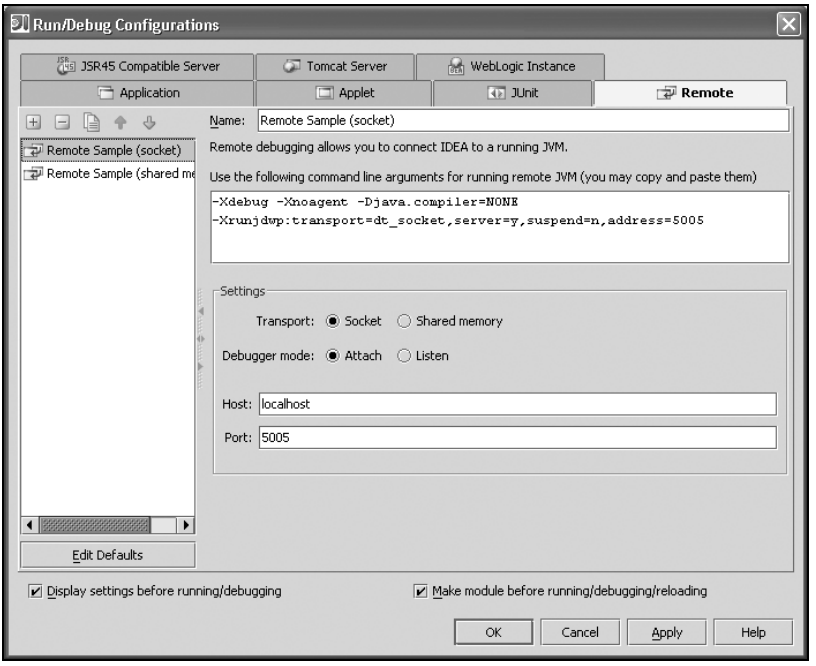


Рис. 9.15. Конфигурация удаленного приложения: соединение через сокет

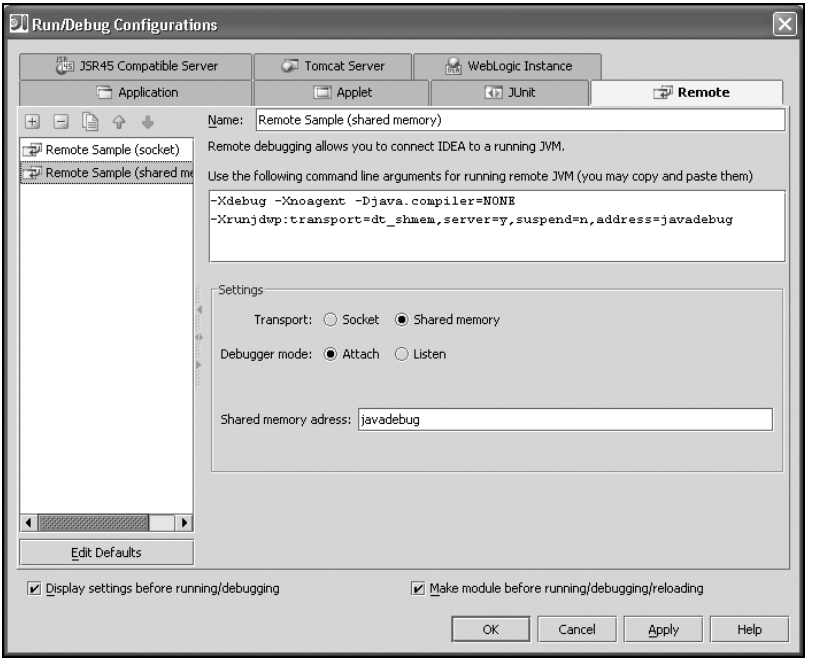


Рис. 9.16. Конфигурация удаленного приложения: соединение через разделяемую память

Среди параметров, которые необходимо использовать для запуска отлаживаемого приложения, имеется параметр `suspend` (приостановить), который может принимать значения "`y`" (yes — да) или "`n`" (no — нет). Если установить значение этого параметра в "`y`", отлаживаемое приложение сразу после запуска приостановит свою работу и будет ждать, когда запустится отладчик.

9.2.5. Web-приложения для Tomcat (закладка *Tomcat Server*)

Общий вид конфигурации Web-приложения для сервера Tomcat представлен на рис. 9.17.

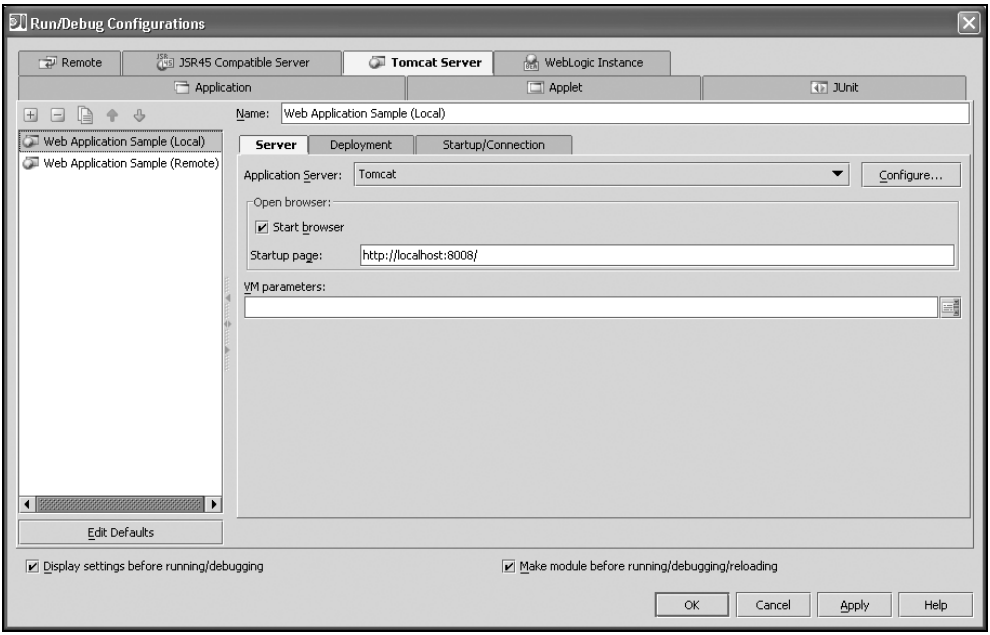


Рис. 9.17. Настройка локальной конфигурации Web-приложения для сервера Tomcat

Имеется два типа конфигурации для сервера Tomcat:

- **Local** (Локальная) (см. рис. 9.17);
- **Remote** (Удаленная) (см. рис. 9.20).

Для обоих типов конфигураций необходимо выбрать из списка сервер приложений (**Application Server**), который будет использоваться для запуска данных конфигураций (*см. раздел 4.1.20*). Для запуска конфигурации Web-приложения для сервера Tomcat для выбора будут доступны только серверы приложений Tomcat. Прочие типы серверов не будут доступны в списке выбора.

9.2.5.1. *Local (Локальная конфигурация)*

Если выбрана локальная конфигурация, то будут доступны три дополнительные закладки со свойствами Web-приложения:

- **Server** (Сервер) — настройки сервера, а также некоторые дополнительные параметры:
  - **Open browser** (Открывать браузер) — открывать окно браузера при старте приложения (**Start browser**) на указанной странице (**Startup page**);
  - **VM parameters** (Параметры VM) — дополнительные параметры виртуальной машины;
- **Deployment** (Развертывание) — выбор модулей, файлы которых будут скопированы в рабочую директорию Web-сервера (рис. 9.18).

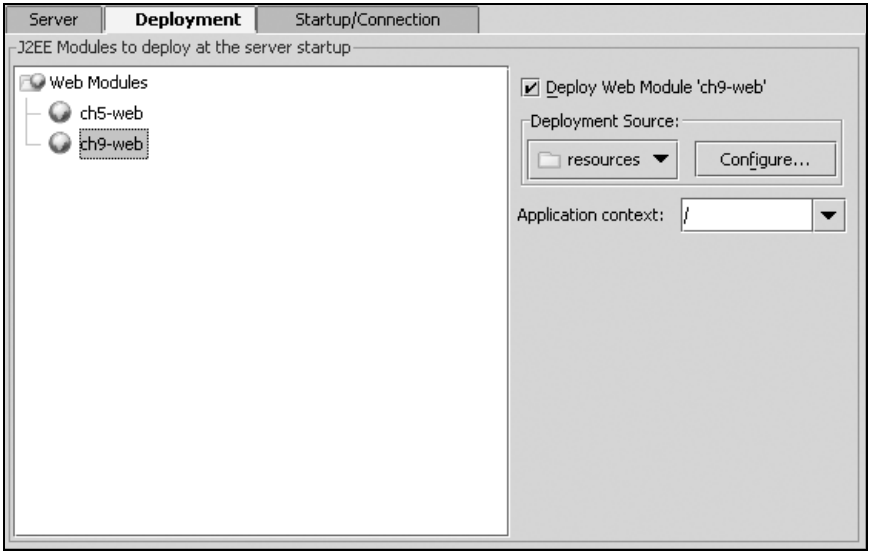


Рис. 9.18. Настройка локальной конфигурации сервера Tomcat: выбор модулей для размещения

Большую часть данной закладки занимает список доступных модулей проекта, настроенных для сервера Tomcat. Для каждого из модулей можно установить:

- **Deploy Web Module** (Опция размещения в рабочей директории Web-сервера). Если данная опция выключена, модуль не будет размещен на сервере;
- **Deployment Source** (Рабочий каталог для размещения файлов модуля);
- **Application context** (Контекст приложения);

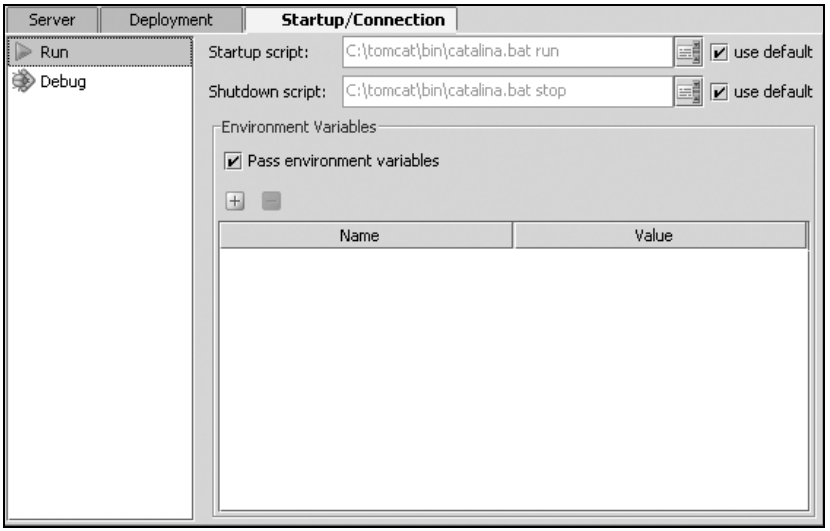


Рис. 9.19. Настройка локальной конфигурации сервера Tomcat: настройка запуска/подсоединения

- **Startup/Connection** (Запуск/Подсоединение) — настройка параметров запуска и подсоединения Web-сервера (рис. 9.19).

  На данной закладке можно настроить следующие параметры для запуска (**Run**) и отладки (**Debug**) Web-приложения:

  - **Startup script** (Программа запуска) — полный путь программы (или командного файла) для запуска Web-сервера. Для данного параметра имеется возможность использовать значение по умолчанию (**use default**);
  - **Shutdown script** (Программа остановки) — полный путь программы (или командного файла) для остановки Web-сервера. Для данного параметра также имеется возможность использовать значение по умолчанию (**use default**);
  - **Pass environment variables** (Передавать переменные окружения) — указание IntelliJ IDEA передавать перечисленные в таблице переменные окружения при запуске Web-сервера.

9.2.5.2. *Remote (Удаленная конфигурация)*

Настройка параметров удаленной конфигурации отличается от локальной отсутствием закладки **Deployment** (Развертывание) (поскольку развертыванием занимается удаленный сервер), а также наличием дополнительных параметров на закладке **Server** (Сервер) (рис. 9.20):

- **Remote Connection Settings** (Настройки удаленного соединения):
  - **Host** (Хост) — название хоста для подсоединения;
  - **Port** (Порт) — номер порта для подсоединения.

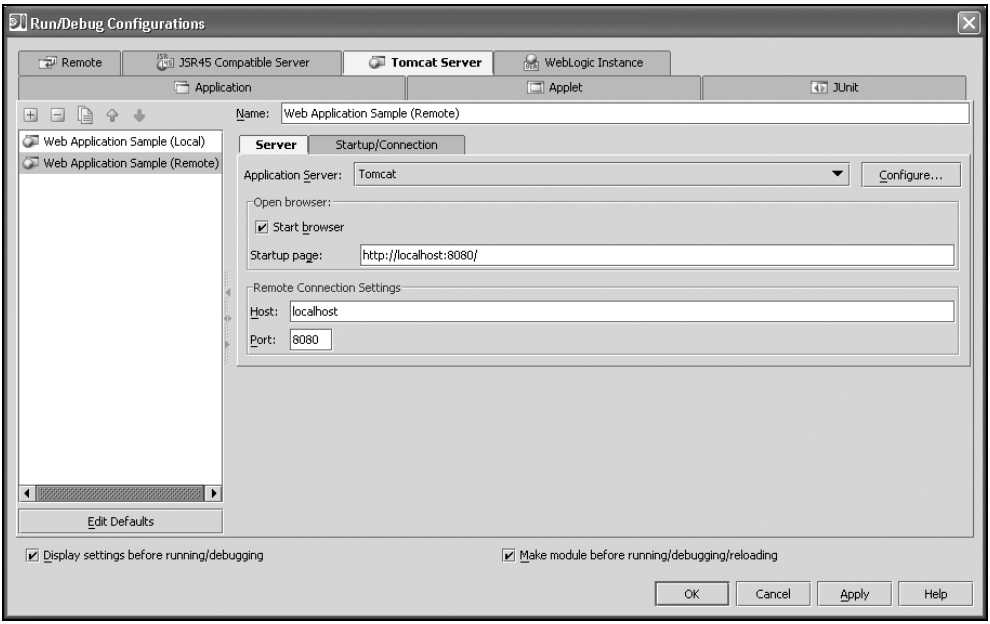


Рис. 9.20. Настройка удаленной конфигурации Web-приложения для сервера Tomcat

Отладчик будет проверять, "жив" ли сервер, посылая запросы по этому адресу.

Параметры в закладке **Startup/Connection** (Запуск/Подсоединение) также несколько отличаются. Параметры запуска (**Run**) вообще отсутствуют, поскольку удаленную конфигурацию сервера Tomcat можно использовать только для отладки, а параметры отладки (**Debug**) представлены в ограниченном виде (рис. 9.21).

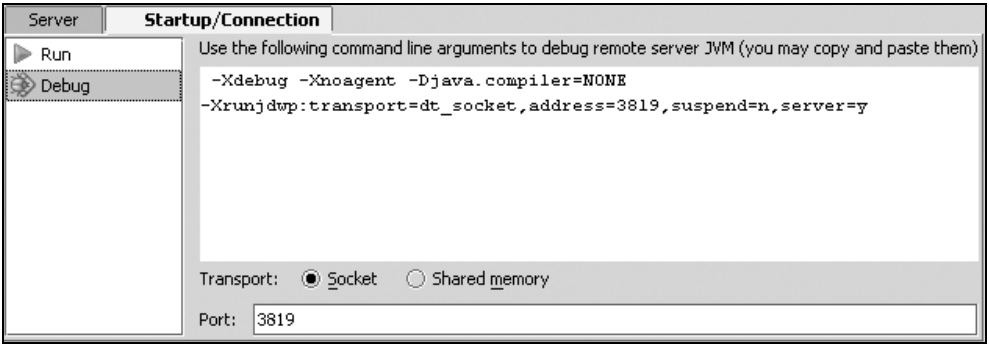


Рис. 9.21. Настройка удаленной конфигурации сервера Tomcat: параметры запуска/подсоединения

На данной закладке указываются только тип протокола (**Transport**), через который будет проходить отладка удаленного приложения, а также номер порта (**Port**). В верхней части закладки отображается информация о параметрах командной строки, с которыми нужно запустить удаленный Web-сервер, чтобы к нему можно было подсоединиться для удаленной отладки.

9.2.6. Приложения для JSR45-совместимых серверов (JSR-45 Compatible Server)

Общий вид конфигурации приложений для серверов, совместимых с JSR45, представлен на рис. 9.22.

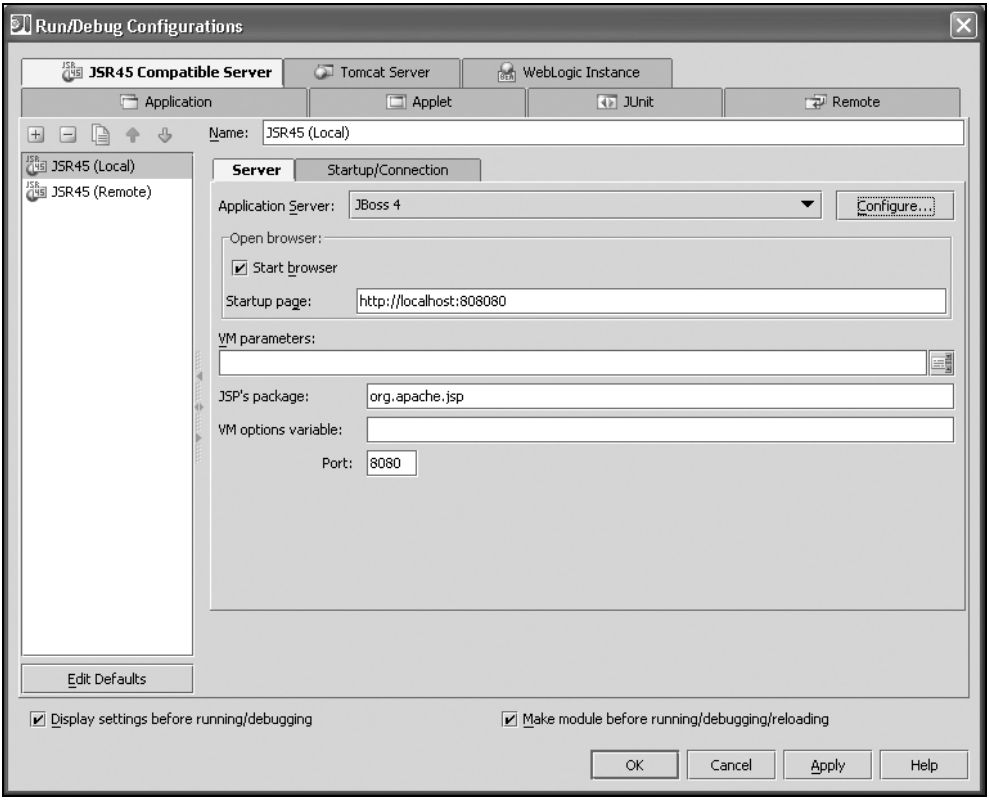


Рис. 9.22. Настройка локальной конфигурации JSR45-совместимого сервера

Имеется два типа конфигурации для JSR45-совместимого сервера:

- **Local** (Локальная) (см. рис. 9.22);
- **Remote** (Удаленная).

Для обоих типов конфигураций необходимо выбрать из списка сервер приложений (**Application Server**), который будет использоваться для запуска данных конфигураций (*см. раздел 4.1.20*). Для запуска конфигурации Web-приложения JSR45-совместимого сервера для выбора будут доступны только JSR45-совместимые серверы. Прочие типы серверов не будут доступны в списке выбора.

9.2.6.1. *Local (Локальная конфигурация)*

Для настройки локальной конфигурации имеются две закладки с параметрами:

- **Server** (Сервер) — настройки сервера, а также некоторые дополнительные параметры:
  - **Open browser** (Открывать браузер) — открывать окно браузера при старте приложения (**Start browser**) на указанной странице (**Startup page**);
  - **VM parameters** (Параметры VM) — дополнительные параметры виртуальной машины;
  - **JSP's package** (Пакет JSP) — название пакета, которому приписываются скомпилированные JSP;
  - **VM options variable** (Переменная с опциями VM) — название переменной, в которую были записаны параметры VM для передачи серверу. Данные переменные можно настроить на закладке **Startup/Connection** (Запуск/Соединение);
  - **Port** (Порт) — номер порта сервера.

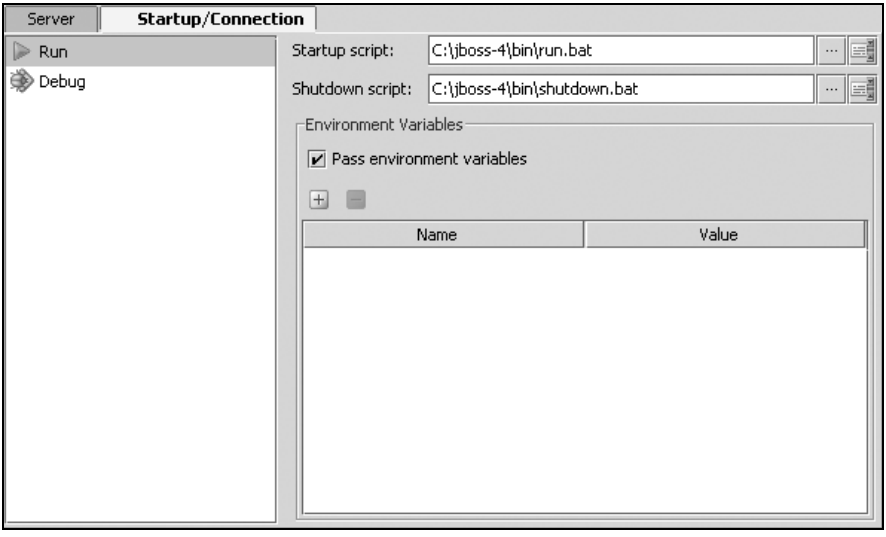


Рис. 9.23. Настройка локальной конфигурации JSR45-совместимого сервера: запуск и соединение

- **Startup/Connection** (Запуск/Подключение) — настройки исполняемых файлов для запуска сервера, а также дополнительные параметры (рис. 9.23).

  На данной закладке можно настроить следующие параметры для запуска (**Run**) и отладки (**Debug**) Web-приложения:

  - **Startup script** (Программа запуска) — полный путь программы (или командного файла) для запуска Web-сервера;
  - **Shutdown script** (Программа остановки) — полный путь программы (или командного файла) для остановки Web-сервера;
  - **Pass environment variables** (Передавать переменные окружения) — указание IntelliJ IDEA передавать перечисленные в таблице переменные окружения при запуске Web-сервера.

9.2.6.2. *Remote (Удаленная конфигурация)*

Настройка удаленной конфигурации практически ничем не отличается от настройки локальной. Единственное исключение — дополнительный параметр **Port** (Номер порта) для удаленной отладки на закладке **Startup/Connection** (Запуск/Соединение) (рис. 9.24).

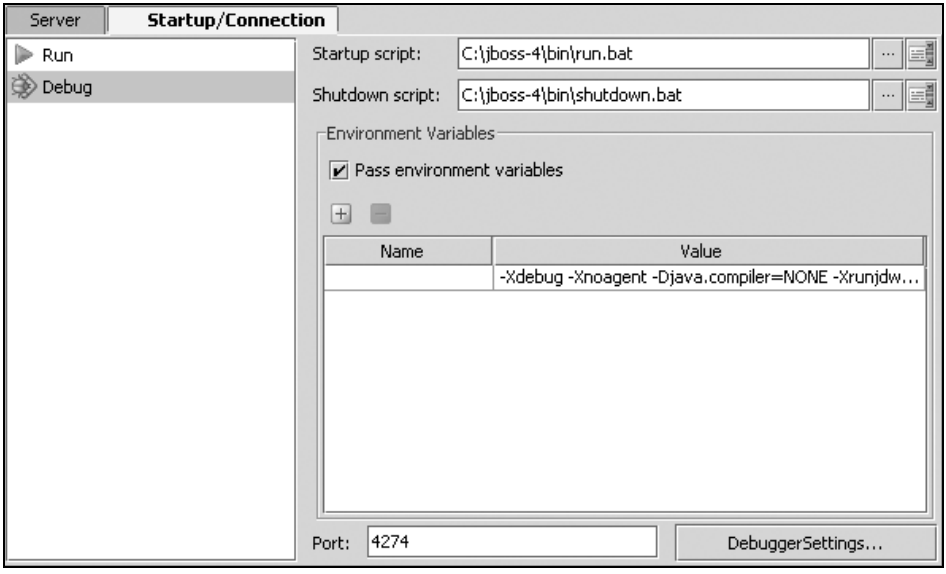


Рис. 9.24. Настройка удаленной конфигурации JSR45-совместимого сервера: запуск и соединение

9.2.7. Приложения для сервера WebLogic (закладка *WebLogic Instance*)

Для того чтобы создать конфигурацию для сервера WebLogic, необходимо, чтобы у вас в проекте уже был модуль, настроенный для использования данного сервера. IntelliJ IDEA может запускать и отлаживать конфигурации как для отдельных Web-модулей и EJB-модулей, так и для полностью настроенных приложений J2EE.

Общий вид конфигурации для серверов WebLogic представлен на рис. 9.25.

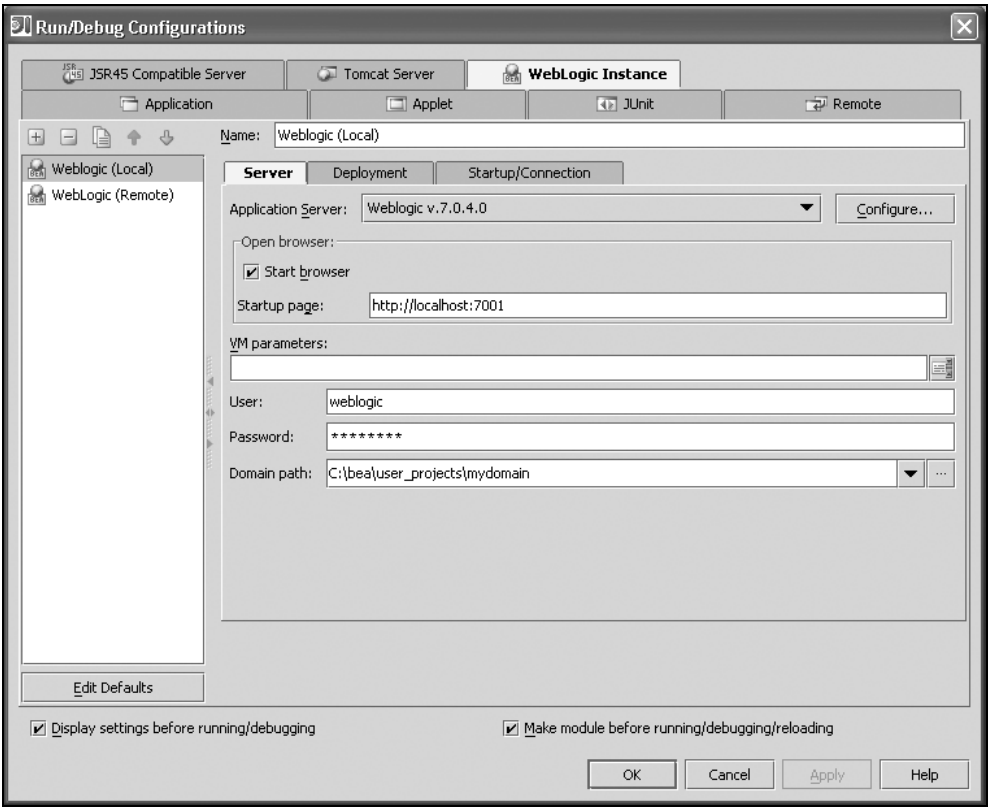


Рис. 9.25. Настройка локальной конфигурации сервера WebLogic

Имеется два типа конфигурации для сервера WebLogic:

- **Local** (Локальная) (рис. 9.25);
- **Remote** (Удаленная).

Для обоих типов конфигураций необходимо выбрать из списка сервер приложений (**Application Server**), который будет использоваться для запуска

данных конфигураций (*см. раздел 4.1.20*). Для запуска конфигурации сервера WebLogic для выбора будут доступны только серверы WebLogic. Прочие типы серверов не будут доступны в списке выбора.

9.2.7.1. *Local (Локальная конфигурация)*

Настройки локальной конфигурации для сервера WebLogic расположены на трех закладках:

- **Server** (Сервер) — настройки сервера, а также некоторые дополнительные параметры:
  - **Open browser** (Открывать браузер) — открывать окно браузера при старте приложения (**Start browser**) на указанной странице (**Startup page**);
  - **VM parameters** (Параметры VM) — дополнительные параметры виртуальной машины;
  - **User** (Пользователь) — учетная запись пользователя для подключения к серверу;
  - **Password** (Пароль) — пароль для подключения к серверу.
  - **Domain path** (Путь домена) — путь директории с доменом. Данную директорию можно выбрать из списка имеющихся директорий, а также выбрать произвольную;
- **Deployment** (Развертывание) — настройки развертывания модулей (рис. 9.26).

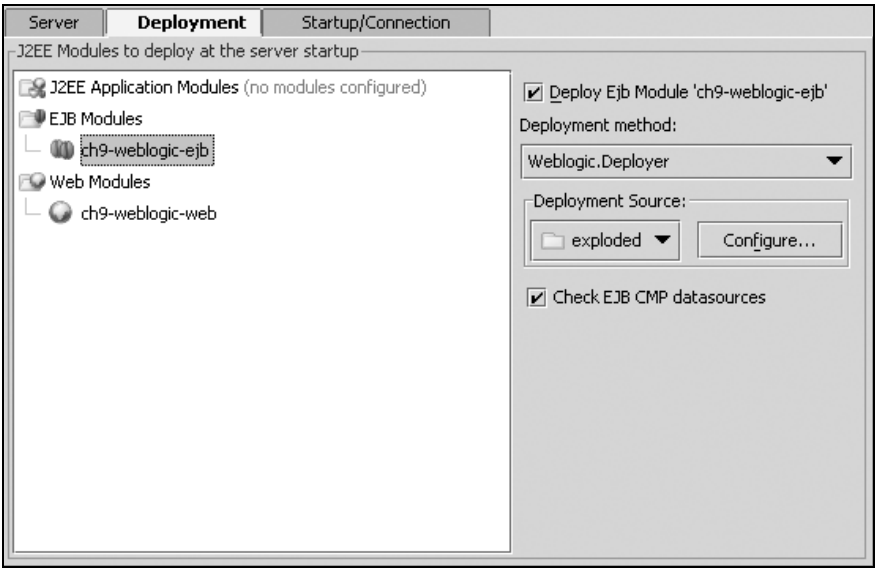


Рис. 9.26. Настройка развертывания модулей для конфигурации сервера WebLogic

Данная закладка содержит список Web-модулей, EJB-модулей, а также модулей приложений J2EE, для которых можно указать следующие опции:

- **Deploy Ejb Module** (Развертывать) — включить развертывание указанного модуля;
- **Deployment method** (Метод развертывания):
- **Weblogic.Deployer** — развертывание с помощью средств WebLogic;
- **Auto deploy** — автоматическое развертывание;
- **Deployment Source** (Источник для развертывания);
- **Check EJB CMP datasources** (Проверять источники данных EJB CMP);

❒ **Startup/Connection** (Запуск/Соединение) — настройки скриптов для запуска сервера (рис. 9.27).

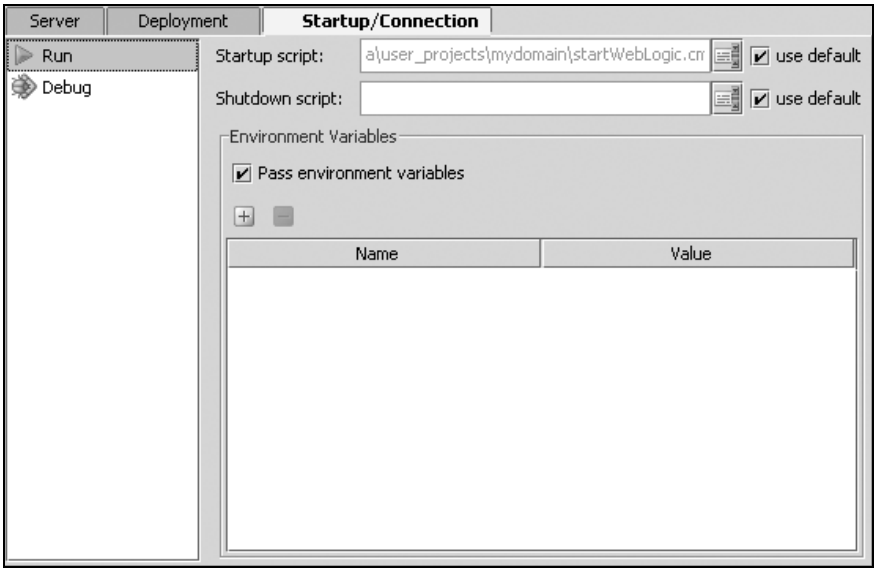


Рис. 9.27. Настройка запуска/соединения конфигурации сервера WebLogic

На данной закладке можно настроить следующие параметры для запуска (**Run**) и отладки (**Debug**) приложения для сервера WebLogic:

- **Startup script** (Программа запуска) — полный путь программы (или командного файла) для запуска Web-сервера. Для данного параметра имеется возможность использовать значение по умолчанию (**use default**);
- **Shutdown script** (Программа остановки) — полный путь программы (или командного файла) для остановки Web-сервера. Для данного параметра также имеется возможность использовать значение по умолчанию (**use default**);

- **Pass environment variables** (Передавать переменные окружения) — указание IntelliJ IDEA передавать перечисленные в таблице переменные окружения при запуске Web-сервера.

9.2.7.2. *Remote (Удаленная конфигурация)*

Общий вид диалога настройки удаленной конфигурации для сервера WebLogic представлен на рис. 9.28.

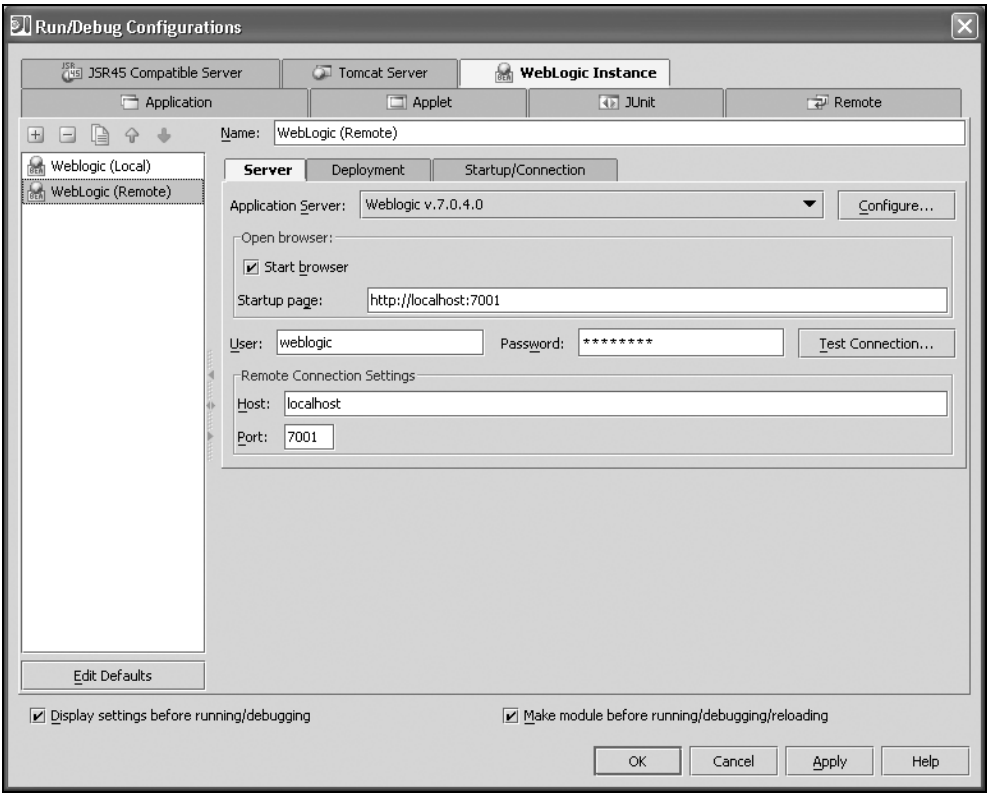


Рис. 9.28. Настройка удаленной конфигурации сервера WebLogic

Опции и параметры удаленной конфигурации также представлены на трех закладках:

❐ **Server** (Сервер) — настройки сервера, а также некоторые дополнительные параметры:

- **Open browser** (Открывать браузер) — открывать окно браузера при старте приложения (**Start browser**) на указанной странице (**Startup page**);
- **VM parameters** (Параметры VM) — дополнительные параметры виртуальной машины;

- **User** (Пользователь) — учетная запись пользователя для подключения к серверу;
- **Password** (Пароль) — пароль для подключения к серверу;
- **Remote Connection Settings** (Настройки удаленного соединения) — адрес хоста (**Host**) и номер порта (**Port**), на котором запущено приложение. Эти данные используются для проверки работоспособности удаленного сервера.

С помощью кнопки **Test Connection** (Проверить соединение) можно протестировать соединение с удаленным сервером, используя указанные параметры.

- **Deployment** (Развертывание) — настройки развертывания модулей. Данные настройки практически идентичны настройкам для локального соединения (*см. раздел 9.2.7.1*), однако в качестве **Deployment Source** (Источника развертывания) должны быть указаны архивы для соответствующих модулей, а не рабочая директория.
- **Startup/Connection** (Запуск/Соединение) — настройки для подсоединения к серверу (рис. 9.29).

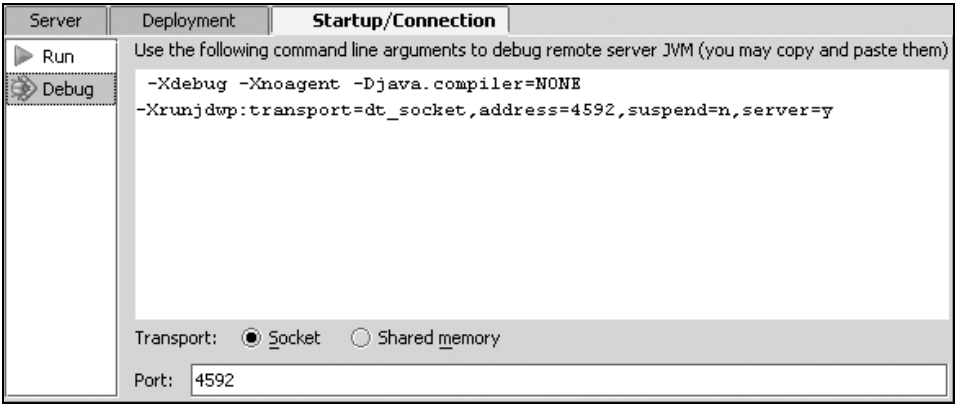


Рис. 9.29. Настройки соединения к удаленному серверу WebLogic

Параметры запуска (**Run**) не имеют настроек, а в параметрах отладки (**Debug**) можно дополнительно указать тип соединения (**Transport**):

- **Socket** (Сокет). Дополнительно имеется возможность задать номер порта (**Port**);
- **Shared memory** (Разделяемая память) — данный тип соединения доступен только при работе под Windows. Дополнительно имеется возможность задать адрес разделяемой памяти (**Shared memory address**).

9.3. Запуск

В IntelliJ IDEA имеется возможность запускать следующие типы Java-приложений:

- ❒ обычные Java-приложения;
- ❒ апплеты;
- ❒ тесты JUnit;
- ❒ Web-приложения, работающие под серверами:
  - Tomcat;
  - JSR45-совместимые серверы;
  - WebLogic.

Рассмотрим запуск перечисленных типов приложений. Для демонстрации возможностей запуска мы создали несколько приложений. Также мы создали конфигурации запуска и отладки для указанных приложений.

9.3.1. Java-приложения (закладка *Application*)

Для примера запуска Java мы создали простой класс, содержащий метод `main()` (листинг 9.1).

Листинг 9.1

```
public class ApplicationSample {
    public static void main(String[] args) {
        System.err.println("Hello, I'm a Java application sample.");
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("  " + (i+1) + ") arg = " + arg);
        }
        System.out.print("Enter some string and press Enter...");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Данное приложение выводит в консоль список параметров, переданных ему при запуске.

Чтобы запустить данное приложение, необходимо в списке доступных конфигураций запуска и отладки выбрать ту конфигурацию, которая соответствует данному приложению (рис. 9.30).

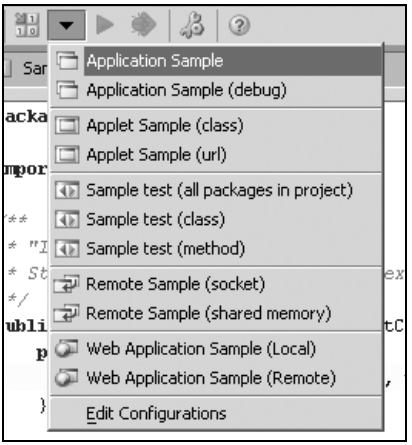


Рис. 9.30. Выбор конфигурации для запуска и отладки

После того как конфигурация выбрана, нужно выполнить команду **Run** (Запуск) (*см. раздел 3.2.10.2*), и приложение будет запущено. Результат работы приложения представлен в консоли, которая отображается в инструментальном окне **Run** (Запуск) (*см. раздел 3.4.8.5*) (рис. 9.31).

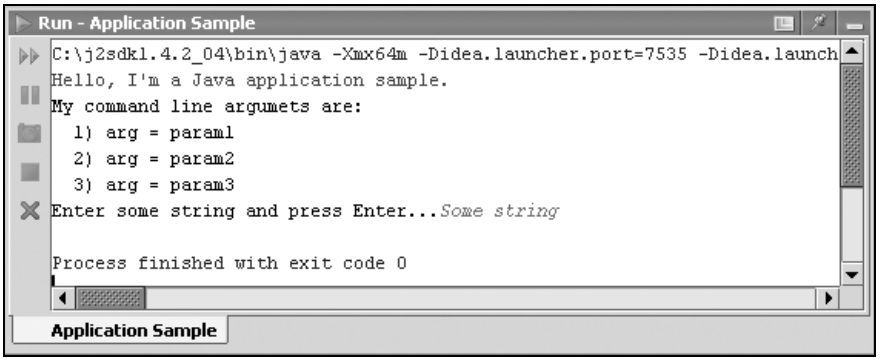


Рис. 9.31. Результат работы приложения в консоли

К сожалению, в черно-белом варианте не видно, что первая и последняя строки в консоли, соответствующие строке запуска приложения и информации о завершении процесса, написаны синим цветом. Это говорит о том, что эти строки вставлены в консоль самой IntelliJ IDEA.

Текст, который был напечатан в консоли приложением, может быть черным или красно-коричневым. Текст печатается черным цветом, если он был выведен через `System.out`, и красно-кирпичным — если был выведен через `System.err`.

Наклонным шрифтом зеленого цвета выделяется текст, который введен в консоль с клавиатуры через `System.in`.

9.3.2. Апплеты

Запуск апплета производится аналогично запуску обычного приложения (*см. раздел 9.3.1*): выбор нужной конфигурации, а затем выполнении команды запуска. Мы будем запускать апплет, представленный в листинге 9.2.

Листинг 9.2

```
public class AppletSample extends JApplet {
    private AppletForm form;

    public static final String [][] MY_PARAMETERS = {
        {"param1", "Some string", "Some parameter 1"},
        {"param2", "Some string", "Some parameter 2"},
        {"param3", "Some string", "Some parameter 3"}
    };

    public String[][] getParameterInfo() {
        return MY_PARAMETERS;
    }

    public void init() {
        super.init();

        form = new AppletForm();
        String paramTable = "<table border=\"1\">";
        for (int i = 0; i < MY_PARAMETERS.length; i++) {
            final String[] myParameter = MY_PARAMETERS[i];
            paramTable += "<tr>";
            for (int j = 0; j < myParameter.length; j++) {
                final String param = myParameter[j];
                paramTable += "<td>" + param + "</td>";
```

```
            }
            paramTable += "</tr>";
        }
        paramTable += "</table>";

        form.setText("<html><body><p>Hello, I'm a Java " +
                     "applet sample. <br>" +
                "My parameters are: </p> " + paramTable);

        getContentPane().add(form);
    }

    class AppletForm extends JPanel {
        private JButton undoButton;
        private JButton doButton;
        private JLabel infoLabel;
        private JPanel mainPanel;

        public AppletForm() {
            add (mainPanel);
            doButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    JOptionPane.showMessageDialog(AppletForm.this,
                            infoLabel.getText());
                }
            });
            undoButton.setEnabled(false);
        }

        public void setText(String text) {
            infoLabel.setText(text);
        }
    }
}
```

Данный апплет применяет форму пользовательского интерфейса, созданную с помощью визуального редактора IntelliJ IDEA (рис. 9.32). Подробнее про формы *см. главу 7.*

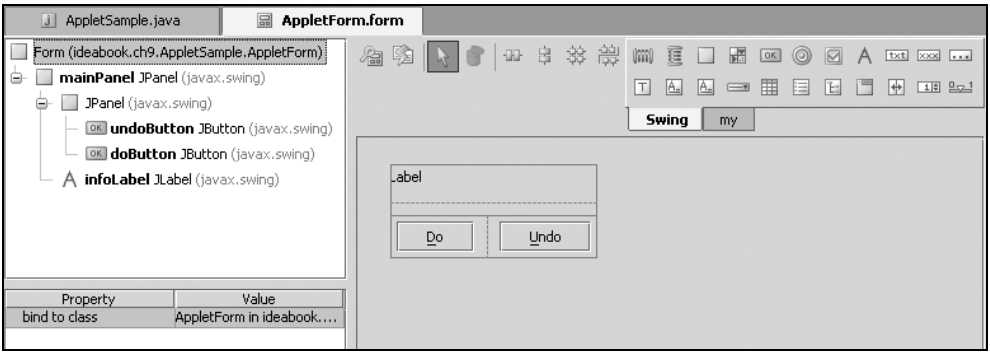


Рис. 9.32. Форма пользовательского интерфейса для апплета

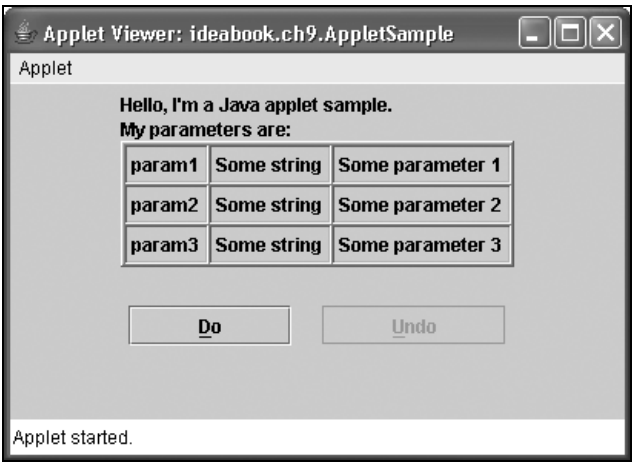


Рис. 9.33. Запуск апплета с помощью утилиты appletviewer

Форма связана с внутренним классом `AppletSample.AppletForm`.

Запуск апплета производится с помощью специальной утилиты appletviewer, входящей в поставку JDK (рис. 9.33).

Для запуска конфигурации апплета, настроенной с использованием HTML-файла, необходим сам HTML-файл. Приводим его текст в листинге 9.3.

Листинг 9.3

```
<html>
<head>
    <title>Java Applet Sample</title>
</head>
```

```
<body>
<applet
    code="ideabook.ch9.AppletSample"
    codebase="./classes"
    width="400" height="300">
    <param name="param1" value="value1"/>
    <param name="param2" value="value2"/>
    <param name="param3" value="value3"/>
    Java applets is not supported in your Internet browser.
</applet>
</body>
```

Запустим наш апплет с помощью HTML-файла из листинга 9.3 в браузере Microsoft Internet Explorer (рис. 9.34).

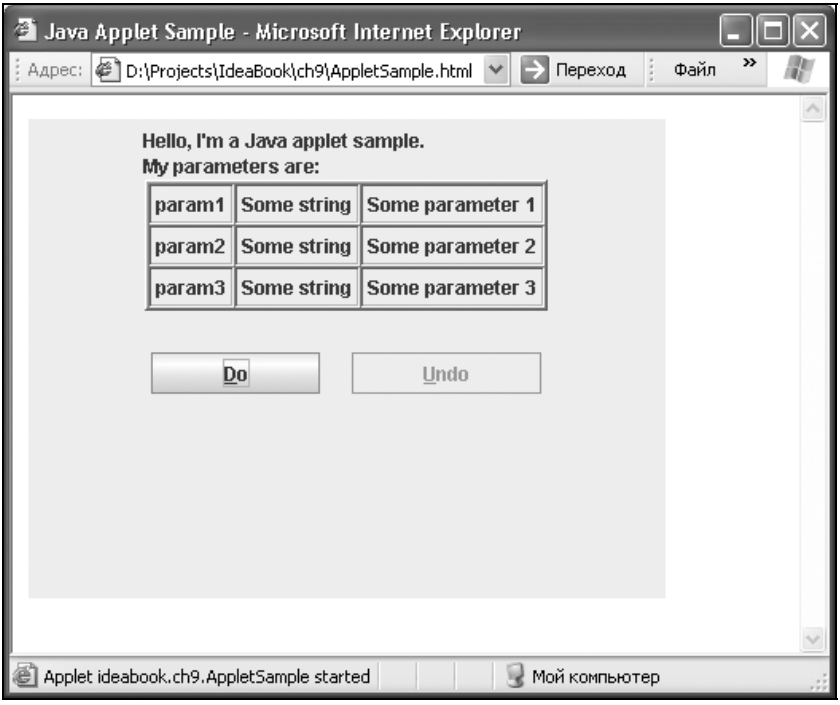


Рис. 9.34. Запуск апплета в браузере

Различия во внешнем представлении кнопок связаны с тем, что в IntelliJ IDEA модуль, в котором находится класс апплета, использует JDK 1.4.2_04, а в браузере используется JRE 5.0.

9.3.3. Тесты JUnit

Для запуска тестов IntelliJ IDEA использует утилиту JUnit. Результаты тестов отображаются в инструментальном окне **Run** (Запуск) (*см. раздел 3.4.8.5*), дополненном специальной формой для учета тестов.

Мы создали два класса с тестами (листинг 9.4).

Листинг 9.4

```
public class SampleTest extends TestCase {
    public void test1 () {
        final String s1 = "the String";
        final String s2 = "the String";
        final String s3 = "tha String";

        assertEquals("Strings [" + s1 + "] and [" + s2 +
                     "] are not equal", s1, s2);
        assertEquals("Strings [" + s2 + "] and [" + s3 +
                     "] are not equal", s2, s3);
    }
    public void test2 () {
        assertTrue("This test is always OK", true);
    }
}

public class SampleTest2 extends TestCase {
    public void testSome () {
        assertTrue("Something wrong", true);
    }
}
```

Для запуска тестов необходимо выбрать конфигурацию тестов, а затем выполнить команду **Run** (Запуск). При старте тестов откроется инструментальное окно **Run** (Запуск) (рис. 9.35).

Обычное содержимое инструментального окна **Run** (Запуск) дополняется двумя панелями, на левой панели отображается иерархический список выполнения тестов, а на правой — информация из консоли тестов (закладка **Output** (Вывод)), а также статистика выполнения тестов (закладка **Statistics** (Статистика)).

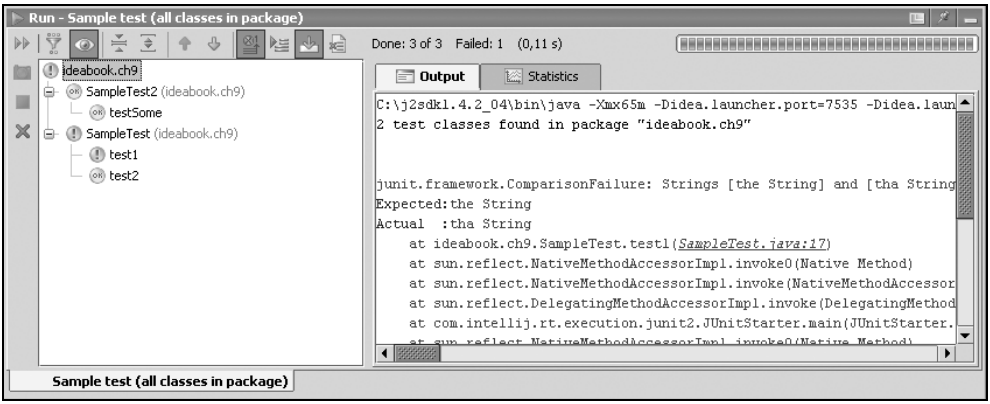


Рис. 9.35. Результаты выполнения тестов в инструментальном окне **Run**

У панелей с результатами работы тестов есть дополнительная панель инструментов, на которой имеются следующие кнопки:

- **Hide Passed** (Прятать прошедшие тесты) — данная кнопка позволяет скрыть из иерархического списка тесты, прошедшие успешно;
- **Track Running Test** (Отслеживать выполняющийся тест) — данная кнопка позволяет указать IntelliJ IDEA перемещать подсветку в иерархическом списке тестов вслед за выполняющимися тестами;
- **Collapse/Expand All** (Свернуть/Развернуть все) (<Ctrl>+<Num Pad +/->) — свернуть/развернуть иерархический список тестов;
- **Next/Previous Failed Test** (Следующий/Предыдущий непрошедший тест) — переход между неудачными тестами;
- **Select First Failed Test When Finished** (Выделить первый непрошедший тест после завершения тестирования);
- **Scroll to Stacktrace** (Прокрутка к стеку вызовов) — прокрутка консоли с результатами тестов к стеку вызовов выделенного теста;
- **Auto Scroll to Source** (Автоматическая прокрутка к исходному тексту теста);
- **Open Source at Exception** (Открывать исходный текст, соответствующий выделенному исключению).

Справа от панели инструментов расположены счетчики запущенных тестов, непрошедших тестов и времени выполнения. В правой верхней части окна отображается указатель прогресса запуска тестов.

На закладке **Statistics** (Статистика) содержится статистическая информация о тестах (рис. 9.36):

- **Time elapsed** (Затраченное время);
- **Usage Delta** (Размер потраченной памяти);

- **Usage Before** (Использовано памяти перед выполнением теста);
- **Usage After** (Использовано памяти после выполнения теста);
- **Results** (Результаты) — результаты выполнения тестов.

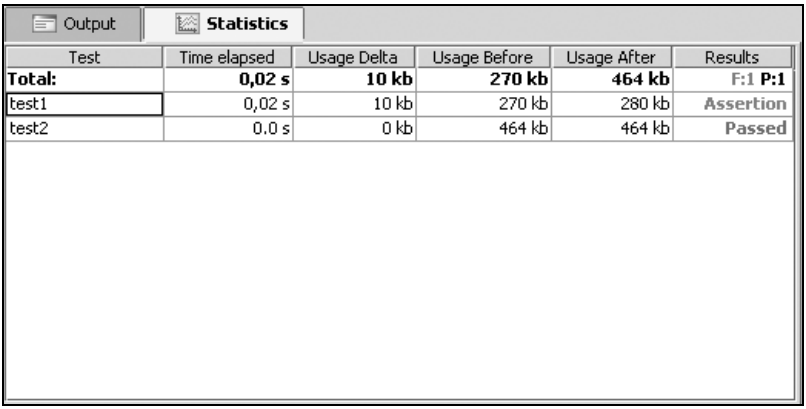


Рис. 9.36. Статистика выполнения тестов

9.3.4. J2EE-приложения

Запуск всех J2EE-приложений, в том числе с конфигурациями для сервера Tomcat, JSR45-совместимого сервера, а также для сервера WebLogic производятся идентично. Мы рассмотрим процесс запуска на примере сервера Tomcat.

Для запуска Web-приложения для сервера Tomcat необходимо выбрать соответствующую конфигурацию (*см. раздел 9.2.5*) и выполнить для него команду **Run** (Запуск).

Мы создали Web-приложение с единственным JSP-файлом index.jsp, содержимое которого приводится в листинге 9.5.

Листинг 9.5

```
<%@ page import="java.util.Map,
                 java.util.Iterator"%>
<%-- This is a web-application sample --%>
<html>
<head>
    <title>Tomcat Server Web-application sample</title>
</head>
<body>
```

```
<p>Hello, I'm a web-application sample.<br/> Here is my parameters: </p>
<%
    if (request.getParameterMap().keySet().size() == 0) {
        %><p>There are no parameters.</p><%
    } else {
%>
<table border="1">
    <thead>
        <tr>
            <th>Name</th><th>Value</th>
        </tr>
    </thead>
    <tbody>
<%
        Map params = request.getParameterMap();

        Iterator iterator = params.keySet().iterator();
        while (iterator.hasNext()) {
            String param = (String) iterator.next();
            String [] values = (String[]) params.get(param);
            %>
        <tr>
            <td><%=param%></td><td><%
            for (int i = 0; i < values.length; i++) {
                String value = values[i];
                %><%=value%> <%
            }
            %></td>
        </tr>
            <%
        }
%>
    </tbody>
</table>
<%
    }
%>
</body>
</html>
```

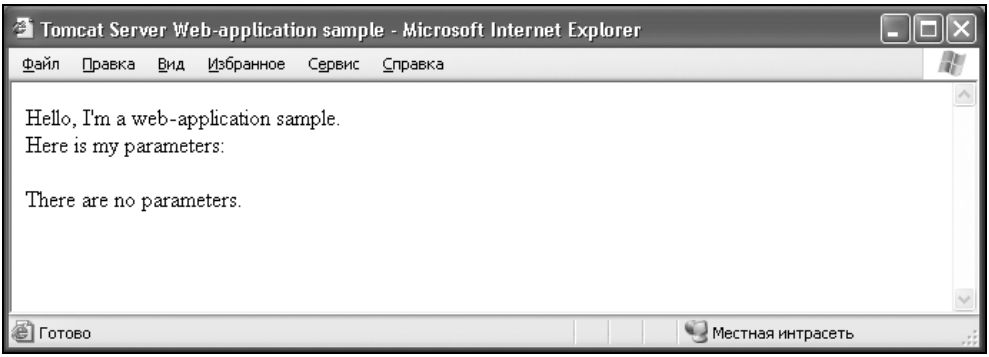


Рис. 9.37. Обращение к Web-приложению из браузера

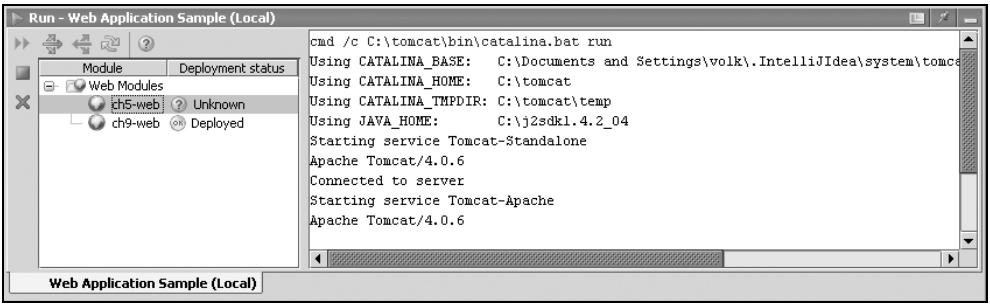


Рис. 9.38. Информация о запуске Web-приложения

Если в конфигурации Web-приложения было указано открывать браузер с соответствующим адресом, он будет открыт (рис. 9.37).

Информация о запуске Web-приложения выводится в инструментальном окне **Run** (Запуск), дополненном двумя информационными панелями (рис. 9.38).

На левой панели выводится список Web-модулей (**Web Modules**) и их статус размещения (**Deployment status**). Модуль может иметь следующие статусы:

- **Deployed** (Размещен) — данный статус имеют модули, файлы которых размещены в рабочей директории Web-сервера;
- **Not deployed** (Не размещен) — данный статус имеют модули, файлы которых не размещены в рабочей директории Web-сервера;
- **Unknown** (Неизвестный) — IntelliJ IDEA не знает, размещены ли файлы данного модуля или нет. Обычно такой статус имеют модули, для которых было выключено автоматическое размещение в настройках конфигурации запуска и отладки Web-приложения (*см. раздел 9.2.5*) и которые не были размещены вручную;

- **Disconnected** (Сервер не подсоединен);
- **Failed** (Неудача) — статус показывает, что развертывание модуля закончилось неудачно.

В правой панели размещается консоль, в которую печатается весь вывод Web-сервера и Web-приложения.

В левой верхней части левой панели имеется дополнительная панель инструментов, на которой содержатся следующие кнопки:

- **Deploy All** (Разместить все) — разместить файлы указанного модуля;
- **Undeploy** (Удалить размещенные файлы) — удалить размещенные файлы указанного модуля. Если директория ресурсов модуля совпадает с рабочей директорией сервера, то физически файлы никуда удалены не будут, приложение продолжит работать, но статус размещения изменится на **Not deployed** (Не размещено). Это небольшая неточность в работе IntelliJ IDEA;
- **Refresh deployment status** (Обновить информацию о размещении) — обновить информацию о статусе размещения. Данная команда сама не производит размещения, а лишь проверяет его.

9.4. Отладка

IntelliJ IDEA использует для отладки приложений библиотеку JPDA фирмы Sun.

IntelliJ IDEA поддерживает отладку следующих типов приложений:

- обычных Java-приложений (Application);
- апплетов (Applet);
- тестов JUnit;
- Web-приложений, работающих под управлением серверов:
  - Tomcat;
  - JSR45-совместимых серверов;
  - WebLogic;
- а также любых Java-приложений, поддерживающих удаленную отладку.

Отладка приложений заключается в возможности производить пошаговое выполнение, вычисление значений локальных переменных и полей класса, установку точек останова (breakpoint) в заданной позиции в коде, на вызов метода, обращение к переменной, а также на выбрасывание заданного типа исключения.

Большинство операций отладки одинаковы для всех типов приложений. Отладка осуществляется с помощью команд отладки, а также посредством инструментального окна **Debug** (Отладка) (*см. раздел 3.4.8.6*).

Далее мы подробно опишем процесс отладки, работу с точками останова и прочими отладочными функциями IntelliJ IDEA.

Мы не будем отдельно акцентировать внимание на отладке того или иного типа приложений за исключением случаев, когда такая отладка имеет отличия от общих правил.

9.4.1. Точки останова

Точки останова — специальным образом помеченные места в коде, на которых происходит остановка выполнения программы.

В IntelliJ IDEA существует четыре типа точек останова:

- строковые точки останова (line breakpoint), активизирующиеся, когда выполнение программы доходит до строки, в которой они стоят;
- точки останова по исключению (exception breakpoint), срабатывающие при возникновении в программе исключения заданного типа;

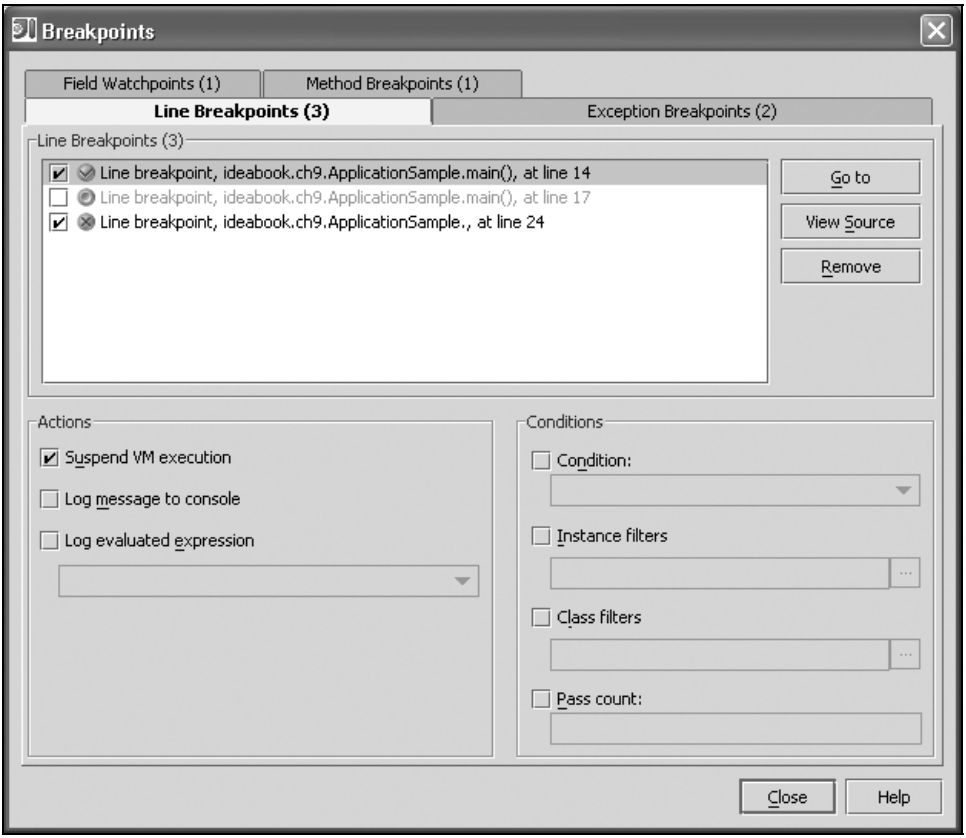


Рис. 9.39. Диалог просмотра и настройки точек останова

- точки останова при вызове метода (method breakpoint), приостанавливающие выполнение программы при вызове определенного метода;
- точки останова при доступе к полю класса (field watchpoint), срабатывающие при обращении к полю класса и/или при изменении его значения.

Каждая из точек останова имеет свое визуальное представление в интерфейсе IntelliJ IDEA. Все точки останова, кроме точек останова по исключению, обозначаются с помощью специальных значков, располагающихся в полях редактора.

Просмотр точек останова, установленных в проекте, а также настройка свойств точек останова производится с помощью диалога **Breakpoints** (Точки останова) (рис. 9.39). Данный диалог можно вызвать либо командой **View Breakpoint** (Просмотр точек останова) (<Ctrl>+<Shift>+<F8>), либо командой **Properties** (Свойства) из контекстного меню точки останова. В данном диалоге имеется четыре закладки — по одной на каждый тип точек останова. В верхней части каждой закладки расположен список установленных точек останова. В начале каждой позиции списка имеется опция отключения точки останова.

9.4.1.1. Общие свойства точек останова

Для каждого типа точек останова имеются дополнительные свойства, которые можно установить персонально для каждой точки. Дополнительные свойства задаются с помощью опций и параметров, разделенных на логические группы, расположенные в нижней части диалога (см. рис. 9.39).

Рассмотрим свойства, общие для всех типов точек останова.

Actions (Действия)

Действия точек останова:

- **Suspend VM execution** (Приостанавливать работу виртуальной машины) — данная опция включена по умолчанию;
- **Log message to console** (Выводить в консоль сообщение) — выводить в консоль отлаживаемого приложения сообщение о достижении точки останова;
- **Log evaluated expression** (Выводить в консоль вычисленное выражение) — при включении данной опции становится доступным поле ввода выражения. В этом поле можно написать любое корректное с точки зрения Java выражение, значение которого будет выведено в консоль. В выражении можно использовать переменные, доступные в текущей области видимости точки останова. Также в этом поле работают все функции редактирования Java-кода, доступные и в обычном редакторе IntelliJ IDEA (*см. раздел 6.1*).

Conditions (Условия)

Дополнительные условия срабатывания точки останова:

- **Condition** (Условие) — непосредственно само условие в формате условия языка Java. В этом поле также работают все функции редактирования Java-кода, доступные и в обычном редакторе IntelliJ IDEA (*см. раздел 6.1*);
- **Instance filters** (Фильтры экземпляров класса) — в данном поле можно ввести ID (идентификаторы) классов — числа формата длинное целое (`long`), обозначающие тот или иной экземпляр класса, в коде которого установлена точка останова. Откуда можно узнать данный идентификатор? Самый простой способ — посмотреть его в инструментальном окне **Debug** (Отладка) на закладке **Frame** (Окно стека) (рис. 9.40).

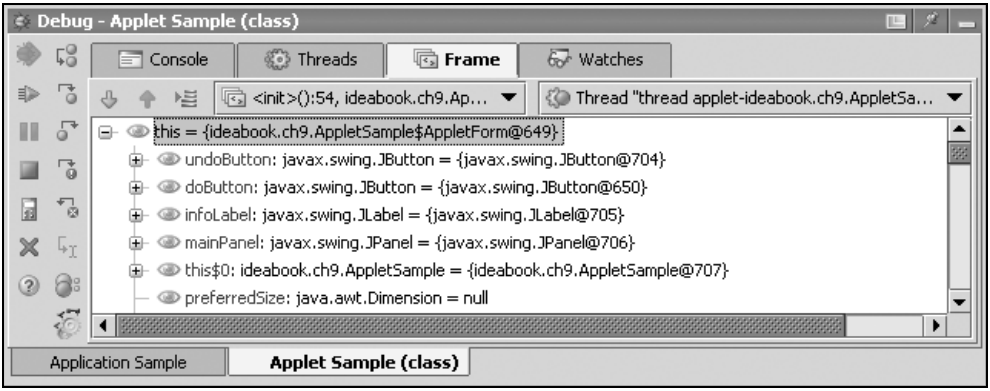


Рис. 9.40. Закладка **Frame** инструментального окна **Debug**

На рис. 9.40 этот ID = 649 — число, написанное справа после имени класса и символа `@`. Если требуется, чтобы точка останова срабатывала для нескольких экземпляров класса, их ID нужно записать через пробел. Для удобства добавления идентификаторов в IntelliJ IDEA предусмотрен специальный диалог, где можно настроить идентификаторы, на которые точка останова будет реагировать (рис. 9.41).

Фильтры можно добавлять (с помощью кнопки **Add** (Добавить)), удалять (с помощью кнопки **Remove** (Удалить)), а также выключать (с помощью выключения соответствующей опции перед номером идентификатора в списке);

- **Class filters** (Фильтры классов). Фильтры классов позволяют задать два типа классов: те, в которых точка останова будет срабатывать, и те, в которых не будет. Для задания классов удобнее всего пользоваться диалогом настройки фильтров классов (рис. 9.42). Этот диалог открывается с помощью кнопки, расположенной справа от поля (см. рис. 9.39).

Рис. 9.41. Настройка фильтров экземпляров класса

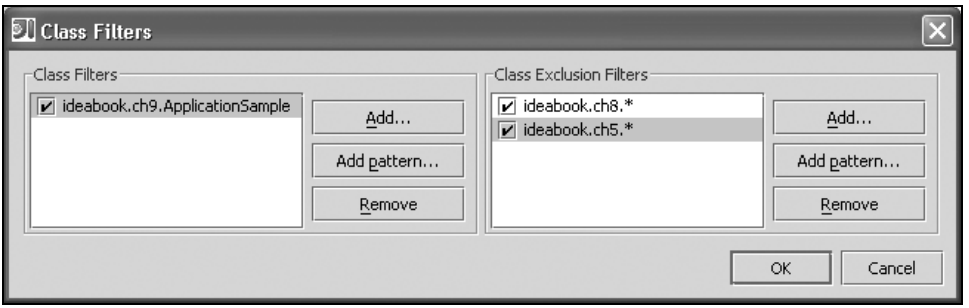


Рис. 9.42. Диалог настройки фильтров классов

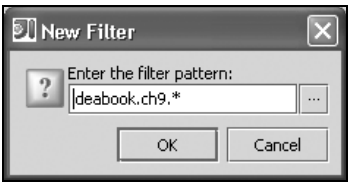


Рис. 9.43. Добавление образца класса

В левой части диалога содержится список классов, которые должны обрабатываться точкой останова (**Class Filters**), в правой — список классов, которые должны исключаться из обработки (**Class Exclusion Filters**). Классы можно добавлять с помощью кнопки **Add** (Добавить). Классы также можно задавать с помощью образца (**Add pattern**) (рис. 9.43).

В образце можно использовать символ '*' для обозначения произвольного класса или пакета. Фильтры классов можно задавать, перечисляя их через пробел. Так, фильтры, которые мы добавили на рис. 9.42, 9.43, можно записать в виде следующей строки:

```
ideabook.ch9.ApplicationSample ideabook.ch9.* -ideabook.ch8.* -
ideabook.ch5.*,
```

где знак '-' перед фильтром обозначает фильтр исключения;

- **Pass count** (Количество проходов) — количество проходов выполнения программы через точку останова, которые будут проигнорированы, т. е. на которых точка не сработает.

Свойства точек останова, характерные для того или иного типа точек, мы подробно рассмотрим в соответствующих *разделах: 9.4.1.2—9.4.1.5.*

9.4.1.2. Строковые точки останова

Строковые точки останова устанавливаются в определенной строке Java-кода, которая имеет однозначное отображение в скомпилированном исполняемом коде.

Если режим отладки не запущен, строковые точки останова выглядят как красный кружок на полях редактора (рис. 9.44).

```
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("  " + (i+1) + ") arg = " + arg);
        }
```

Рис. 9.44. Строковая точка останова

```
    public static void main(String[] args) {
        System.err.println("Hello, I'm a Java application sample.");
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("  " + (i+1) + ") arg = " + arg);
        }
        System.out.print("Enter some string and press Enter...");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
Line breakpoint, ideabook.ch9.ApplicationSample., at line 24
Breakpoint is invalid : No executable code found at line 24 in class ideabook.ch9.ApplicationSample.
Suspend VM: yes
Log message: no
```

Рис. 9.45. Строковые точки останова: отладка запущена

Если запустить отладку, точки останова изменят свой вид в зависимости от того, в допустимом месте они установлены или нет (рис. 9.45).

На рис. 9.45 изображены (сверху вниз):

- точка останова, установленная в неправильном месте (помечена крестиком);

- точка останова, установленная в правильном месте (помечена галочкой);
- выключенная точка останова (помечена зеленым вогнутым кружочком);
- точка останова, установленная в неправильном месте (помечена крестиком).

Если подвести курсор мыши к значку точки останова на полях, во всплывающем окне будет отображена информация о ней. Например, так, как это показано на рис. 9.45 для последней точки останова.

Если щелкнуть значок точки останова правой кнопкой мыши, откроется контекстное меню из трех пунктов:

- **Disable** (Выключить) — сделать точку останова неактивной, но не убирать ее;
- **Remove** (Удалить) — удалить точку останова;
- **Properties** (Свойства) — открыть диалог настройки точек останова.

Строковые точки останова не имеют специфических свойств, кроме тех, что описаны в *разделе 9.4.1.1.* Рассмотрим примеры установки некоторых параметров, например, параметра **Log evaluated expression** (Выводить в консоль вычисленное выражение) из группы **Actions** (Действия) (рис. 9.46).

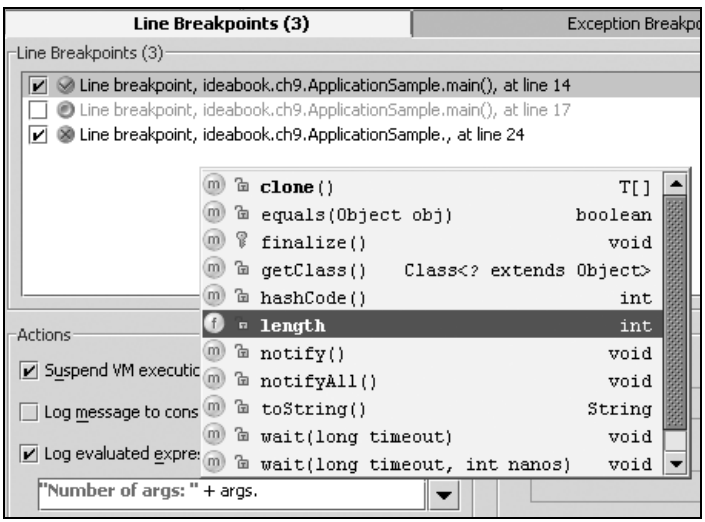


Рис. 9.46. Редактирование выражения для вывода в консоль при достижении точки останова

Мы ввели в качестве значения выражения строку `"Number of args: " + args.length`, в результате чего при достижении точки останова в консоль было выведено сообщение (рис. 9.47).

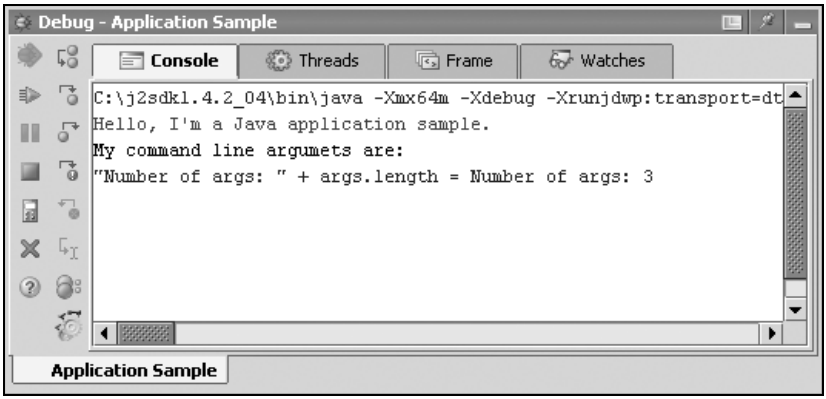


Рис. 9.47. Сообщение при достижении точки останова

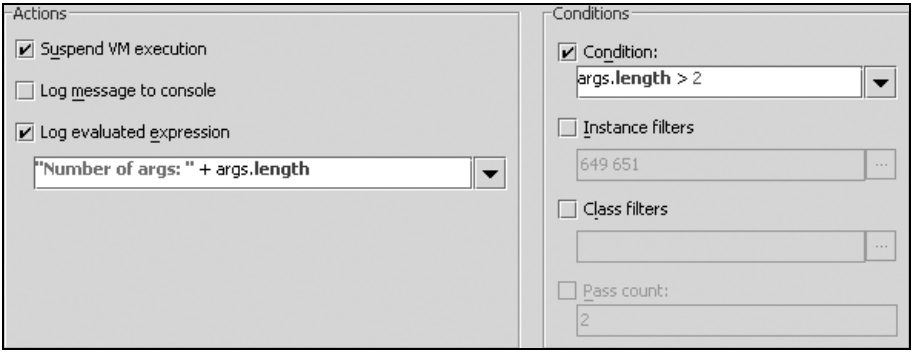


Рис. 9.48. Пример дополнительных параметров строковой точки останова

Для параметра **Condition** (Условие) из группы **Conditions** (Условия) мы ввели `args.length > 2`, чтобы точка останова срабатывала только при количестве аргументов большем двух. Все свойства указанной точки останова представлены на рис. 9.48.

9.4.1.3. Точки останова по исключению

Точки останова по исключению не имеют визуального представления в редакторе до тех пор, пока они не будут активированы, т. е. пока в процессе отладки не возникнет соответствующее исключение.

Точки останова по исключению довольно удобно ставить, например, на исключение `NullPointerException`, чтобы иметь возможность проследить окружение, вызвавшее эту точку останова.

Точки останова устанавливаются с помощью диалога просмотра и настройки точек останова, который может быть открыт командой **Run | View Breakpoint** (Запуск | Просмотр точек останова) (рис. 9.49).

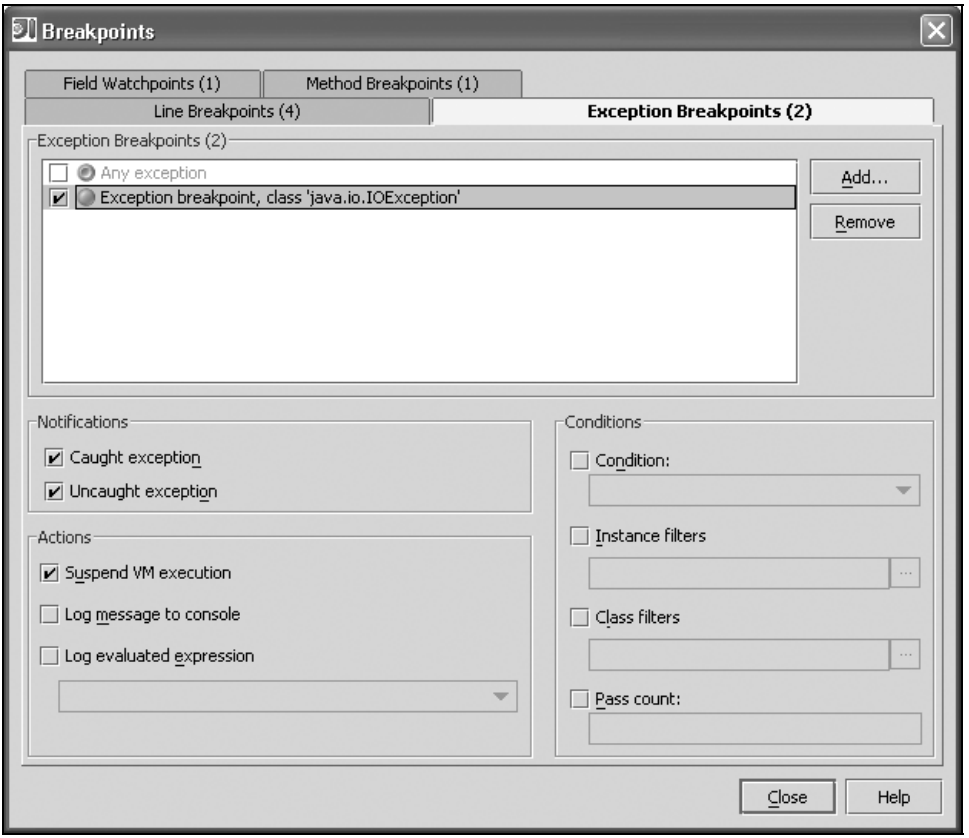


Рис. 9.49. Настройка точек останова по исключению

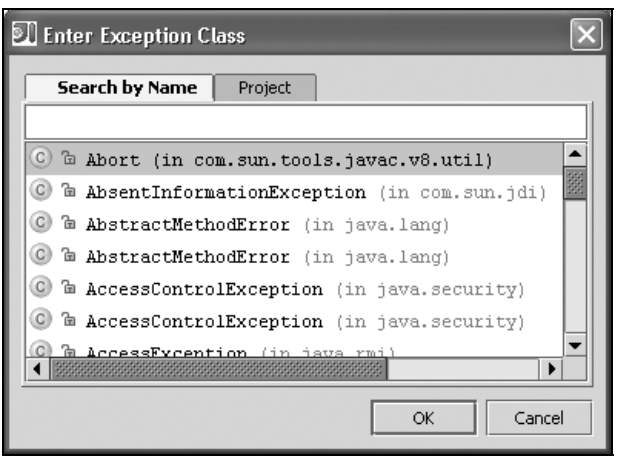


Рис. 9.50. Выбор класса-исключения

В верхней части закладки **Exception Breakpoints** (Точки останова по исключению) содержится список установленных точек останова. Добавить новые точки можно с помощью кнопки **Add** (Добавить), которая откроет диалог выбора класса (рис. 9.50), а удалить с помощью кнопки **Remove** (Удалить).

Точки останова по исключению имеют дополнительную группу опций (см. рис. 9.49).

Notifications (Уведомления)

Уведомления позволяют указать дополнительные условия срабатывания точки останова:

- **Caught exception** (Обработанное исключение) — учитывать исключения, для которых существует блок `catch`;
- **Uncaught exception** (Необработанное исключение) — учитывать исключения, для которых не задан блок `catch`.

9.4.1.4. Точки останова при вызове метода

Точки останова при вызове метода позволяют приостанавливать выполнение приложения в случае вызова метода, на котором стоит точка останова.

Визуально точки останова при вызове метода обозначаются красным кружком с четырьмя черными точками на полях редактора (рис. 9.51).

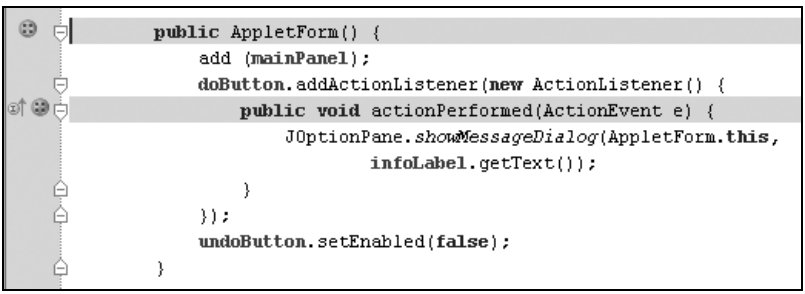


Рис. 9.51. Точки останова при вызове метода

На рис. 9.51 представлены две точки останова при вызове метода, первая находится в выключенном состоянии (на черно-белой иллюстрации это не очень заметно, но в ее центре расположен зеленый вогнутый кружок), а вторая — в рабочем.

При запуске отладки те точки, которые располагаются в корректных местах кода и будут обработаны при вызове метода, помечаются галочкой (рис. 9.52).

Точки останова при вызове метода имеют дополнительную группу опций (рис. 9.53).

```
        public AppletForm() {
            add (mainPanel);
            doButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    JOptionPane.showMessageDialog(AppletForm.this,
                            infoLabel.getText());
                }
            });
            undoButton.setEnabled(false);
        }
```

Рис. 9.52. Точки останова при вызове метода: отладка запущена

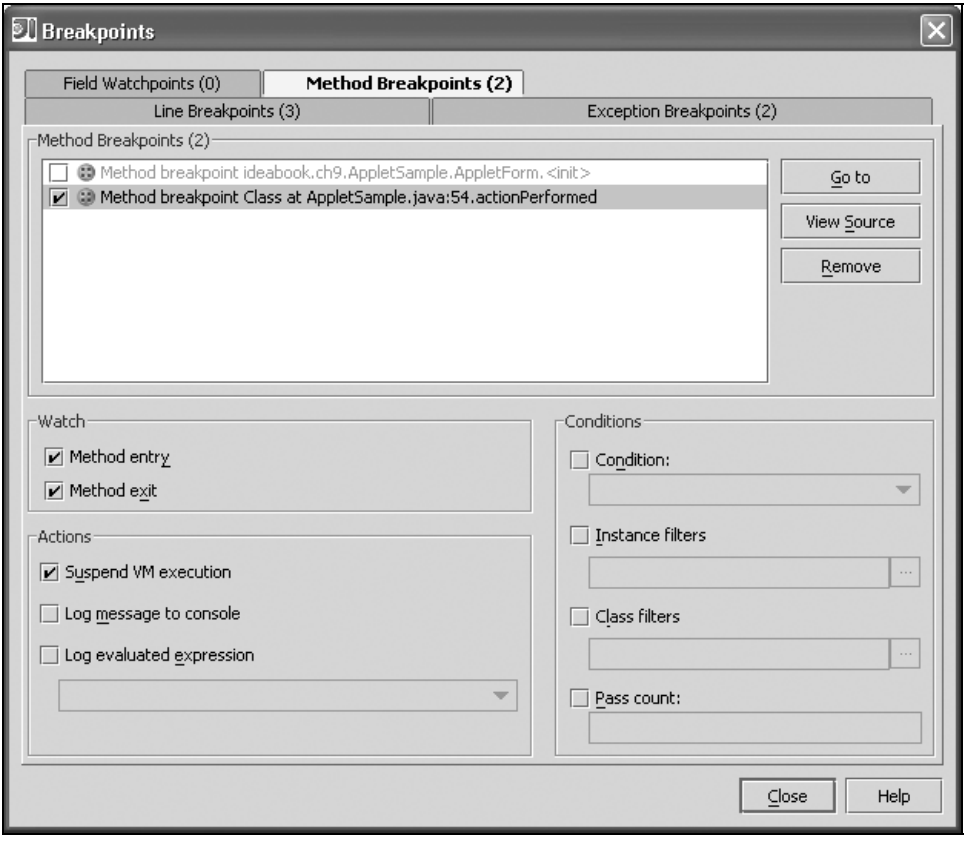


Рис. 9.53. Точки останова при вызове метода

Watch (Наблюдение)

Данная группа настроек позволяет указать две дополнительные опции:

- **Method entry** (Выход в метод) — точка останова должна срабатывать при входе в метод;
- **Method exit** (Выход из метода) — точка останова должна срабатывать при выходе из метода.

9.4.1.5. Точки останова при доступе к полю класса

Точки останова при доступе к полю класса срабатывают при обращении к тому или иному полю класса. Такие точки отображаются в виде красного кружка с черной черточкой (рис. 9.54).

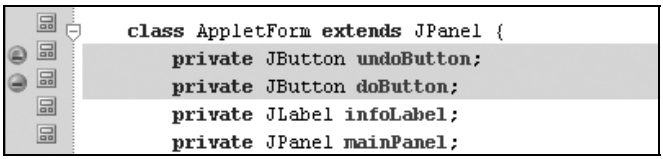


Рис. 9.54. Точки останова при доступе к полю класса

На рис. 9.54 изображены: выключенная точка останова при доступе к полю (на черно-белой иллюстрации это не очень заметно, но в ее центре расположен зеленый вогнутый кружок), точка останова в рабочем состоянии.

При включенном режиме отладки точки останова приобретают специальные пометки в виде галочки, свидетельствующие о том, что точка останова активна и сработает при обращении к полю (рис. 9.55).

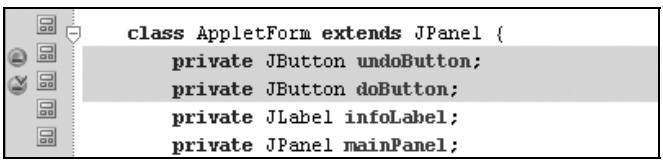


Рис. 9.55. Точки останова при доступе к полю класса: отладка запущена

Кроме стандартных свойств, точки останова при доступе к полю класса имеют дополнительную группу свойств (рис. 9.56).

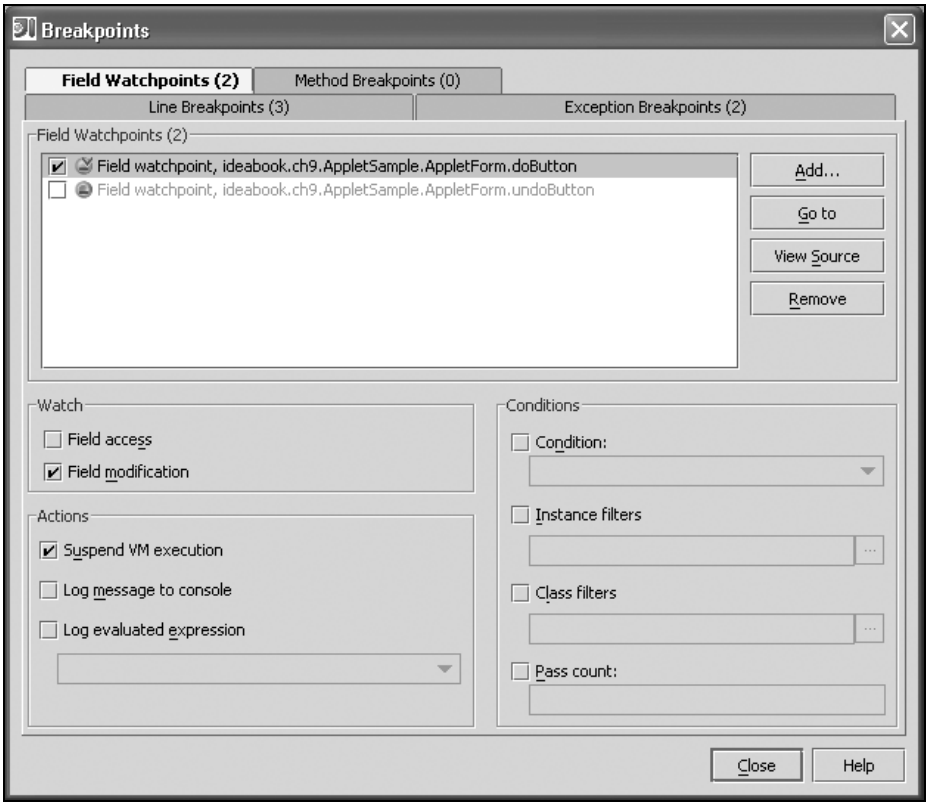


Рис. 9.56. Точки останова при доступе к полю класса

Watch (Наблюдение)

Данная группа настроек позволяет указать две дополнительные опции:

- **Field access** (Доступ к полю) — точка останова будет срабатывать при любом обращении к полю;
- **Field modification** (Изменение поля) — точка останова будет срабатывать только в случае изменения значения поля.

9.4.2. Пошаговая отладка

Пошаговая отладка выполняется с помощью специальных команд, которые позволяют выполнять последовательно все операторы, встречающиеся в программе. Команды для пошаговой отладки представлены в главном меню IntelliJ IDEA в разделе **Run** (Запуск). Всего есть шесть команд, для отладки программы в пошаговом режиме:

- **Step Over** (Шаг через) — один шаг выполнения программы без захода в вызовы методов;

- **Step Into** (Шаг внутрь) — один шаг выполнения программы с заходом в вызываемые методы;
- **Step Out** (Шаг наружу) — один шаг на один уровень выше;
- **Force Step Into** (Принудительный шаг внутрь) — один шаг выполнения программы с принудительным заходом в вызываемые методы;
- **Run to Cursor** (Выполнение до курсора) — выполнение программы до курсора;
- **Pop Frame** ("Вытолкнуть" окно стека) — перемещение к месту вызова текущего метода.

Далее мы подробно рассмотрим пошаговую отладку с использованием всех этих команд.

Но для начала нам нужно запустить отладку программы, для чего выберем интересующую нас конфигурацию и выполним команду **Debug** (Отладка) <Shift>+<F9>.

Затем, чтобы перейти в пошаговый режим, необходимо, чтобы программа либо остановилась сама на какой-нибудь точке останова, либо мы должны приостановить ее с помощью команды **Pause Program** (Приостановить программу) из меню **Run** (Запуск).

Поставим точку останова в первую строку метода `main` приложения из листинга 9.1 и запустим отладку. Выполнение программы тут же остановится на указанной точке останова (рис. 9.57).

```
public class ApplicationSample {
    public static void main(String[] args) {
        System.err.println("Hello, I'm a Java application sample.");
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("  " + (i+1) + ") arg = " + arg);
        }
        System.out.print("Enter some string and press Enter...");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Рис. 9.57. Выполнение программы приостановлено в точке останова

Строка, выделенная темно-синим фоном, указывает на текущее место выполнения. Продолжим отладку с использованием пошаговых команд.

9.4.2.1. *Step Over (Шаг через)*

"Горячая" клавиша: <F8>.

Выбрав команду **Step Over** (Шаг через), мы выполним первую строку приложения и переместимся ко второй строке, при этом в консоль будет напечатано то, что выводит первая строка (рис. 9.58).

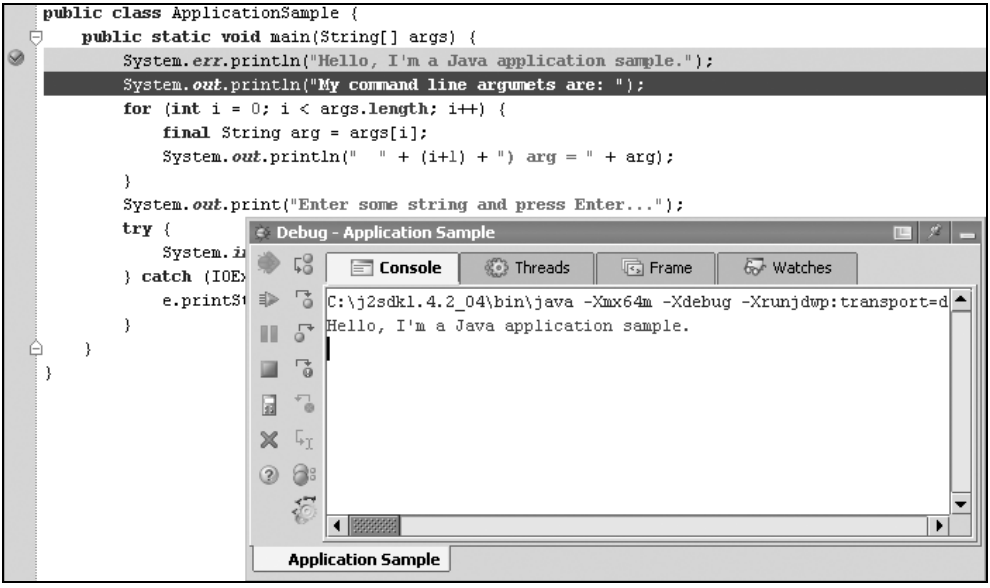


Рис. 9.58. Команда **Step Over** выполнена

9.4.2.2. *Step Into (Шаг внутрь)*

"Горячая" клавиша: <F7>.

Если мы сейчас выполним команду **Step Into** (Шаг внутрь), это совсем не означает, что мы окажемся внутри метода `println()`, как того следует ожидать. Скорее всего, мы просто перейдем к следующей строке нашего приложения. Давайте выполним команду и проверим. Да, действительно, выполнение программы перешло к следующей команде без захода внутрь метода `println()`. Почему так вышло? Дело в том, что IntelliJ IDEA имеет настройки отладчика, которые по умолчанию указывают не входить в методы, относящиеся к стандартным классам поставки Java (из пакетов com.sun.*, java.*, javax.*, org.omg.*, sun.*, а также junit.*). Эти настройки можно изменить с помощью диалога **Debugger** (Отладчик) в общих настройках IntelliJ IDEA (*см. раздел 4.1.9*).

Существует специальная команда пошагового выполнения, которая позволяет входить в такие методы принудительно (*см. раздел 9.4.2.4*).

```
public class ApplicationSample {
    public static void main(String[] args) {
        System.err.println("Hello, I'm a Java application sample.");
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("   " + (i+1) + ") arg = " + arg);
        }
        System.out.print("Enter some string and press Enter...");
```

Рис. 9.59. Программа остановилась еще на одном вызове метода `println()`

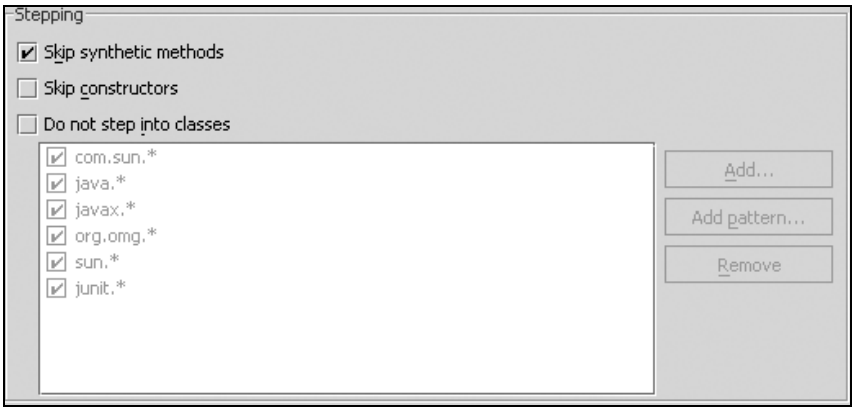


Рис. 9.60. Запрет, не входить в методы стандартных классов, выключен

```
    // This method is invoked by the virtual machine to load a class.
    private synchronized Class loadClassInternal(String name)
    throws ClassNotFoundException
    {
    return loadClass(name);
    }
```

Рис. 9.61. Команда **Step Into** привела нас внутрь метода загрузчика классов

Пока же выполним еще два "шага внутрь" и остановимся еще на одном вызове метода `println()` (рис. 9.59).

Теперь в настройке отладчика уберем запрет входить в методы стандартных классов (рис. 9.60), и снова выполним команду **Step Into** (Шаг внутрь).

И вот мы оказались внутри вызова метода из загрузчика классов (рис. 9.61).

В большинстве случаев нам действительно не нужно лезть внутрь системных классов, поэтому лучше снова включить запрет на вход в такие классы.

9.4.2.3. *Step Out (Шаг наружу)*

"Горячие" клавиши: <Shift>+<F8>.

Выбраться из системного кода обратно в наше приложение нам поможет команда **Step Out** (Шаг наружу). Данная команда переносит выполнение к следующему оператору в коде, расположенному по стеку вызова методов на один уровень выше. Это похоже на то, как если бы на предыдущем шаге мы выполнили команду **Step Over** (Шаг через) вместо **Step Into** (Шаг внутрь).

Выполнив эту команду (может быть понадобится выполнить команду несколько раз, это зависит от версии JDK, которую вы используете), мы снова окажемся в нашем приложении внутри цикла `for`.

Выполним с помощью команды **Step Over** (Шаг через) несколько витков цикла, пока он не кончится.

9.4.2.4. *Force Step Into (Принудительный шаг внутрь)*

"Горячие" клавиши: <Shift>+<F7>.

Данная команда позволяет выполнить шаг выполнения программы с принудительным заходом во все методы, ограничения на которые можно установить в диалоге настройки отладчика (*см. раздел 4.1.9*):

- синтетические методы (методы, сгенерированные компилятором);
- конструкторы;
- классы, перечисленные в списке исключенных классов.

Выполнение нашей программы остановилось в очередной строчке с вызовом метода `println()` (рис. 9.62).

```
public class ApplicationSample {
    public static void main(String[] args) {
        System.err.println("Hello, I'm a Java application sample.");
        System.out.println("My command line argumets are: ");
        for (int i = 0; i < args.length; i++) {
            final String arg = args[i];
            System.out.println("  " + (i+1) + ") arg = " + arg);
        }
        System.out.print("Enter some string and press Enter...");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Рис. 9.62. Выполнение программы остановилось на очередном вызове метода `println()`

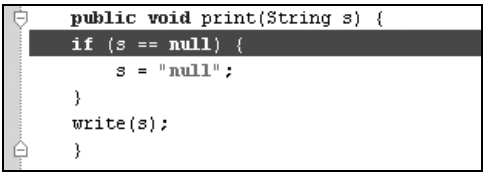


Рис. 9.63. Команда пошагового выполнения с принудительным входом внутрь вызовов методов

В прошлый раз (*см. раздел 9.4.2.2*) нам удалось войти в такой метод, только включив ограничения в диалоге настройки отладчика. Теперь с помощью команды **Force Step Into** (Принудительный шаг внутрь) мы сможем войти внутрь данного метода без изменения настроек (рис. 9.63).

9.4.2.5. *Run to Cursor (Выполнение до курсора)*

"Горячие" клавиши: <Alt>+<F9>.

Данная команда играет роль своеобразной временной точки останова. Если поместить курсор в некоторой строке кода и выбрать команду, программа будет выполняться до тех пор, пока ее выполнение не дойдет до указанной строки.

Если в промежутке между вызовом команды и остановкой выполнения программы в указанной строке сработают какие-либо точки останова, выполнение программы будет приостановлено.

Поместим курсор в строке вызова метода `read()` и выполним команду **Run to Cursor** (Выполнение до курсора) (рис. 9.64).

Выполнение программы остановится в указанной строке (рис. 9.65).

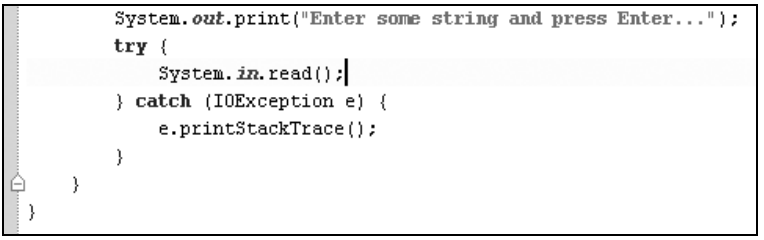


Рис. 9.64. Выполнение программы до указанной строки

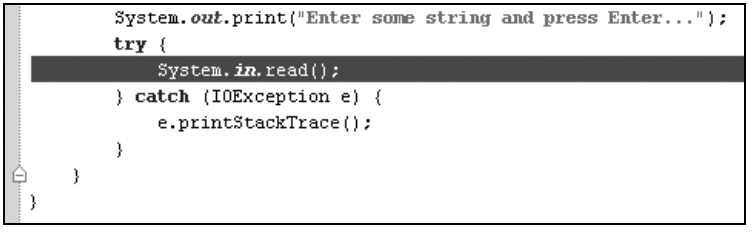


Рис. 9.65. Выполнение программы остановилось в указанной строке

9.4.2.6. *Resume Program (Продолжить выполнение программы)*

Данная команда также часто используется при пошаговой отладке приложений. Ее нужно выполнить, если дальнейшая пошаговая отладка больше не требуется.

Данная команда продолжит выполнение программы так, как будто она и не останавливалась.

Выполним ее и закончим отладку нашего приложения.

9.4.2.7. *Pop Frame ("Вытолкнуть" окно стека)*

Данная команда позволяет "отменять" вызов какого-либо метода. Ее выбор приводит к тому, что стек вызова метода сбрасывается, а выполнение программы оказывается в точке вызова метода.

Для демонстрации работы данной команды рассмотрим отладку приложения, представленного в листинге 9.6.

Листинг 9.6

```
public class MathSample {
    private boolean sumCalled = false;

    public int doSum (int a1, int a2, int a3) {
        System.out.println(sumCalled ?
                           "doSum() already called" :
                           "doSum() yet not called");
        sumCalled = true;
        return a1 + a2 + a3;
    }

    public static void main(String[] args) {
        final MathSample sample = new MathSample();

        final int sum = sample.doSum(1, 2, 3);
        System.out.println("Sum (1 + 2 + 3) = " + sum);
    }
}
```

Поставим точку останова в последнюю строку метода `doSum()` (строка с оператором `return`) и запустим отладку приложения (рис. 9.66).

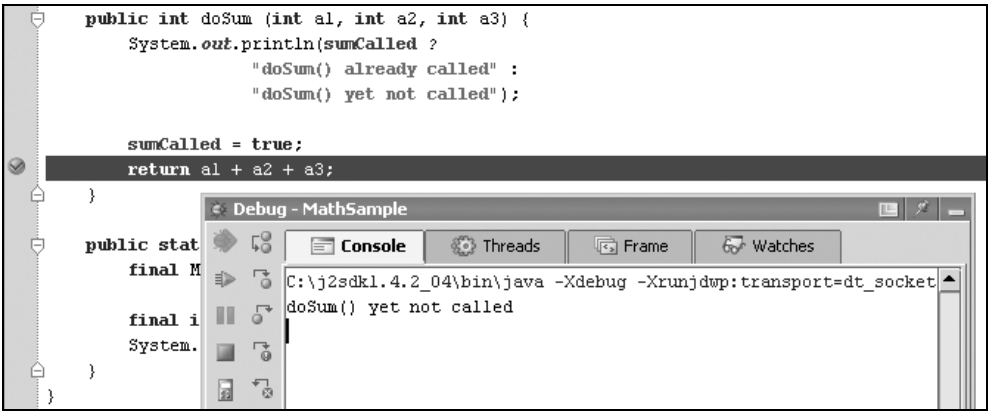


Рис. 9.66. Выполнение программы остановилось в точке останова

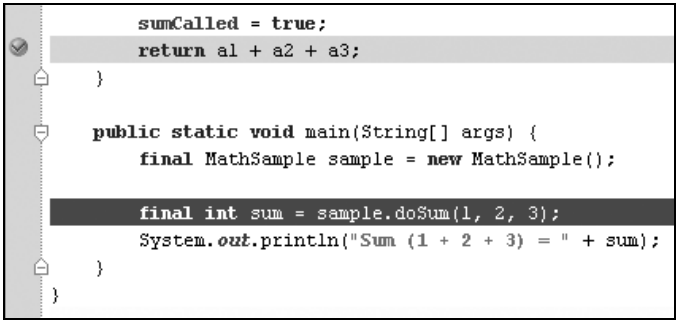


Рис. 9.67. Результат выполнения команды **Pop Frame**

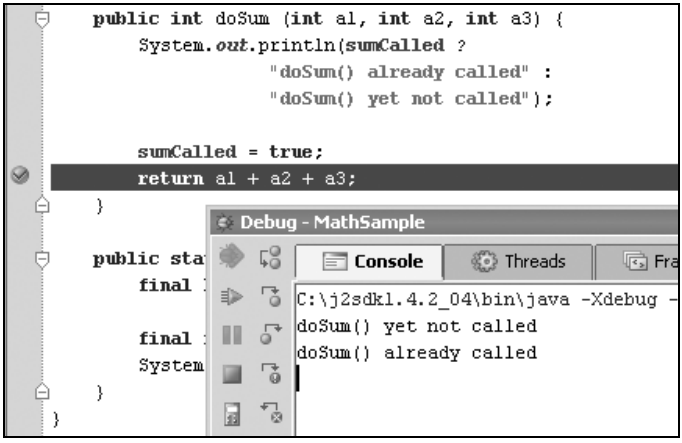


Рис. 9.68. Результат повторного выполнения метода `doSum()`

При этом в консоли приложения будет напечатана строка "`doSum() yet not called`", поскольку флаг `sumCalled` содержал значение `false`.

Теперь выполним команду **Pop Frame** ("Вытолкнуть" окно стека), выполнение программы вернется к месту вызова метода (рис. 9.67).

Теперь выберем команду **Resume Program** (Продолжить выполнение программы) (*см. раздел 9.4.2.6*), и выполнение программы снова остановится на точке останова (рис. 9.68).

Как можно заметить, в консоли было напечатано сообщение "`doSum() already called`", что свидетельствует о том, что флаг `sumCalled` равен `true`. Это значение было присвоено ему при предыдущем вызове метода `doSum()`, который мы "откатили".

Данная команда производит "откат" выполнения метода, но она не исправляет всех изменений, внесенных методом, поэтому следует быть особо внимательным при работе с этой командой.

9.4.3. Информация о потоках (закладка *Threads*)

Во время отладки имеется возможность просматривать информацию о потоках, работающих в рамках отлаживаемого приложения. Данная информация представлена на закладке **Threads** (Потоки) инструментального окна **Debug** (Отладка) (рис. 9.69).

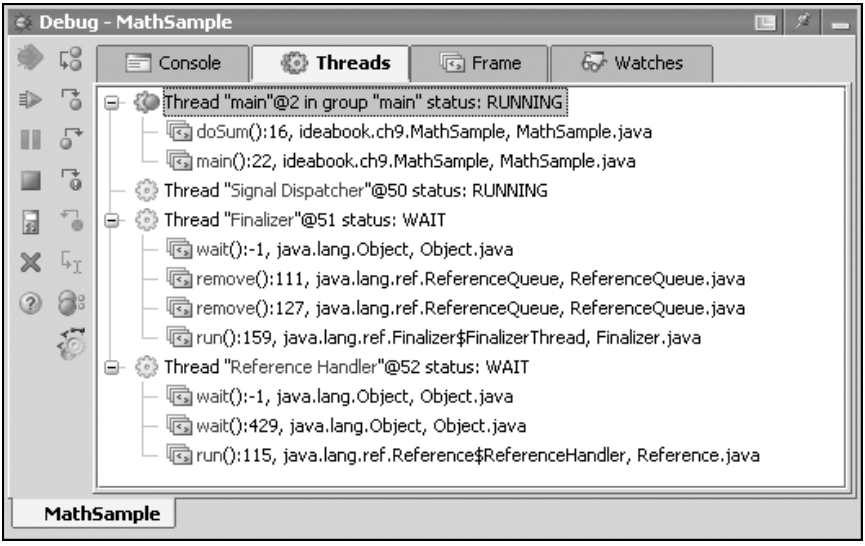


Рис. 9.69. Информация о потоках в приложении

Кроме самих потоков, в данном окне приводится информация о стеке вызова методов в потоке, а также о текущем состоянии потока:

- `RUNNING` (Поток выполняется);
- `WAIT` (Поток ожидает);
- `MONITOR` (Поток ожидает на Java-мониторе);
- `NOT_STARTED` (Поток создан, но не запущен);
- `SLEEPING` (Поток "спит");
- `UNKNOWN` (Состояние неизвестно);
- `ZOMBIE` (Поток завершил свое выполнение);
- `UNDEFINED` (Состояние не определено).

В списке потоков имеется контекстное меню со следующими командами:

- **Resume** (Продолжить) — продолжить выполнение выбранного потока;
- **Freeze** (Приостановить) — приостановить выполнение выбранного потока;
- **Export Threads** (Экспортировать потоки) — данная команда дублирует команду из меню **Run** (Запуск) (*см. раздел 3.2.10.21*);
- **Customize View** (Настройка отображения) — изменение представления информации о потоках. Данная команда открывает диалог, представленный на рис. 9.70.

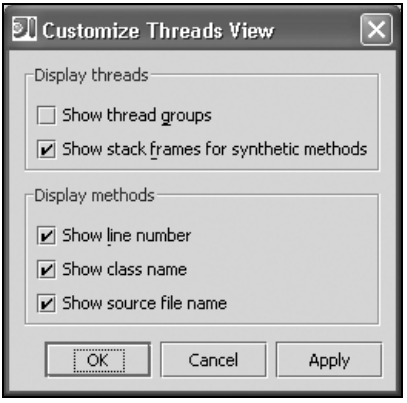


Рис. 9.70. Диалог настройки отображения потоков

С помощью данного диалога можно настроить следующие опции отображения потоков:

- **Display threads** (Отображение потоков) — опции настройки отображения потоков:
- **Show thread groups** (Показывать группы потоков);

- **Show stack frames for synthetic methods** (Показывать окна стека для синтетических методов — методов, сгенерированных компилятором);
- **Display methods** (Отображение методов) — опции настройки отображения методов:
- **Show line number** (Показывать номера строк);
- **Show class name** (Показывать имя класса);
- **Show source file name** (Показывать имя файла с исходными текстами класса).

Дополнительно для методов в стеке вызова потока в контекстном меню имеются следующие команды:

- **Show Frame** (Показать окно стека) — команда отображает информацию о переменных, доступных в области видимости указанного метода из стека вызова методов. Информация отображается на закладке **Frame** (Окно стека) инструментального окна **Debug** (Отладка) (*см. раздел 9.4.4*);
- **Pop Frame** ("Вытолкнуть" окно стека) — команда дублирует команду из меню **Run** (Запуск) (*см. раздел 9.4.2.7*);
- **Jump to Source** (Переход к исходному тексту) (<F4>) — команда открывает в редакторе исходные тексты класса, в котором находится метод из стека потока.

9.4.4. Информация о переменных (закладка *Frame*)

Информация о переменных и их значениях, находящихся в области видимости данной точки выполнения программы (окна стека вызова методов), доступна на закладке **Frame** (Окно стека) инструментального окна **Debug** (Отладка) (рис. 9.71).

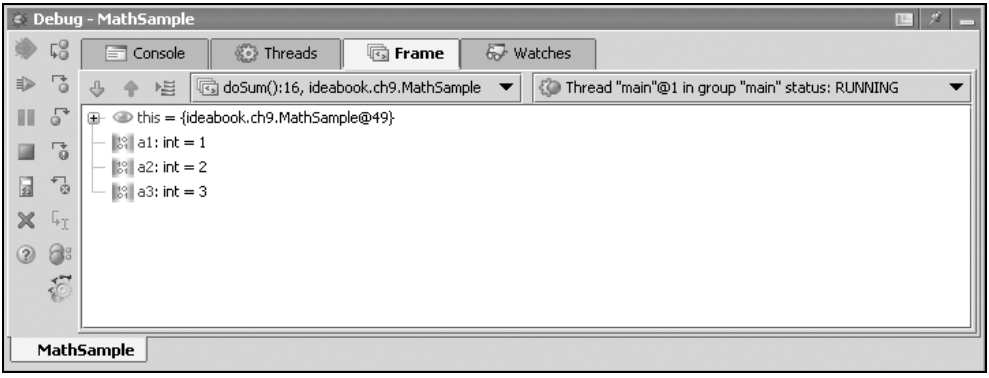


Рис. 9.71. Информация о переменных и их значениях

В верхней части закладки расположена панель инструментов с тремя кнопками:

- **Previous Frame** (Предыдущее окно стека) (<Ctrl>+<Alt>+<Down>) — перемещение к предыдущему окну с переменными из предыдущего метода в стеке;
- **Next Frame** (Следующее окно стека) (<Ctrl>+<Alt>+<Up>) — перемещение к следующему окну с переменными из следующего метода в стеке;
- **Show Execution Point** (Показать точку выполнения) (<Ctrl>+<F10>) — команда, дублирующая команду из меню **Run** (Запуск) (*см. раздел 3.2.10.16*).

Также в верхней части закладки расположены два выпадающих списка:

- список со стеком вызовов методов текущим потоком (именно по этому стеку вызовов и позволяют перемещаться команды **Previous/Next Frame** (Предыдущее/Следующее окно));
- список потоков, работающих в приложении — данный список совпадает с информацией, представленной на закладке **Threads** (Потоки) (*см. раздел 9.4.3*).

Переменные в списке представлены в иерархическом виде, т. е. все объекты выводятся в виде списка полей, а те, в свою очередь, — в виде списка своих полей и т. д.

В данном списке переменных работает контекстное меню со следующими командами:

- **Inspect** (Инспектировать) — команда открывает окно, в котором отображает информацию о переменной (рис. 9.72).

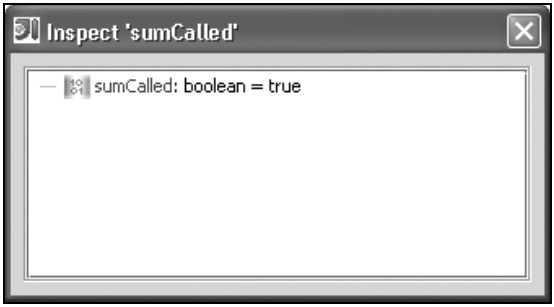


Рис. 9.72. Окно инспектирования значения переменной

  В отдельном окне гораздо удобнее инспектировать изменения значения переменной по мере отладки;
- **Set value** (Установить значение) — команда позволяет установить значение переменной (рис. 9.73);

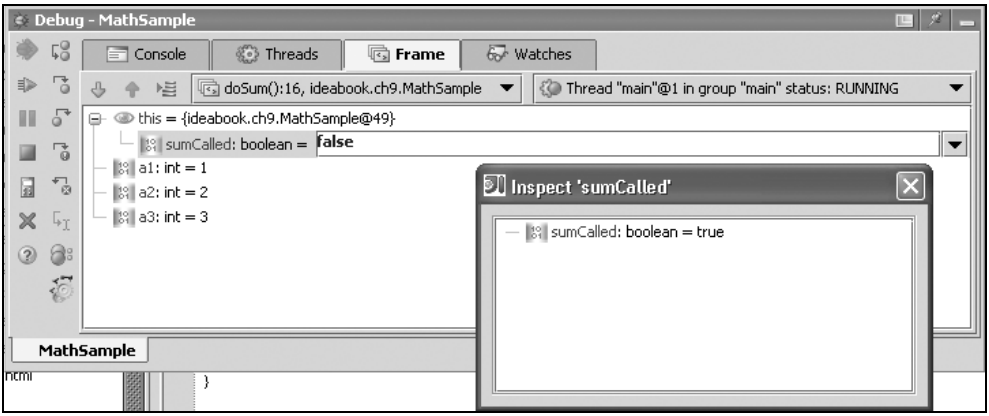


Рис. 9.73. Изменение значения переменной

- **Jump to Source** (Переход к исходному тексту) (<F4>) — команда открывает в редакторе исходные тексты класса, в котором определена данная переменная;
- **Add to Watches** (Добавить в окно наблюдений) — команда позволяет добавить переменную в окно наблюдаемых значений (**Watches**) (*см. раздел 9.4.5*);
- **Add Field Watchpoint** (Добавить точку останова при доступе к полю класса) — добавить точку останова при доступе к полю класса для выбранного поля класса;
- **Copy Value** (Копировать значение);
- **Customize View** (Настроить отображение) — команда открывает диалог с настройками отображения информации о переменных (рис. 9.74). В данном диалоге представлены следующие опции:
  - **Sort alphabetically** (Сортировать по алфавиту) — сортировать переменные в списке по алфавиту. Если опция выключена, переменные располагаются в том порядке, в котором они объявлены в коде;
  - **Select new locals in scope** — прокручивать окно с переменными так, чтобы были видны новые переменные;
  - **Alternate view for Collections classes** (Альтернативный способ просмотра информации о коллекциях) — если данная опция включена, содержимое коллекции будет представлено в виде последовательного списка элементов (рис. 9.75, *а*). Если опцию выключить — коллекции будут отображаться так же, как и все другие классы (рис. 9.75, *б*);
  - **Arrays length limit** (Лимит длины массива) — максимальное количество элементов массива, отображающееся в окне просмотра информации;
  - **Show static fields** (Показывать статические поля);

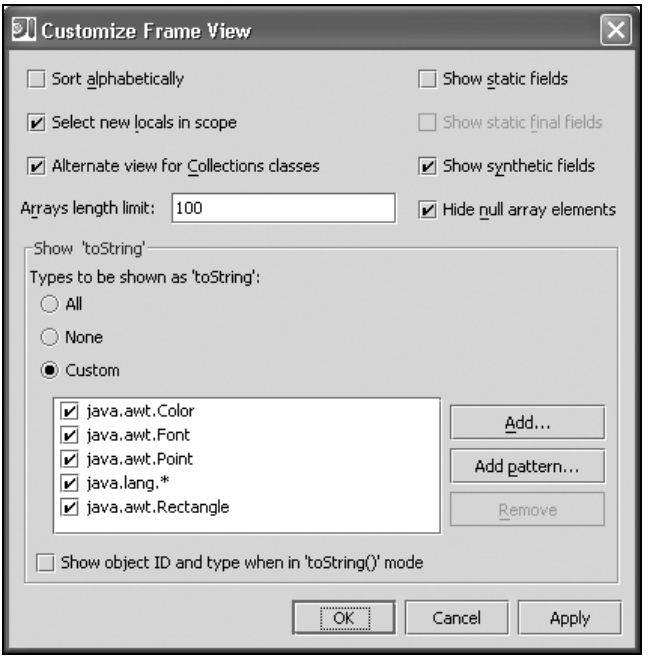


Рис. 9.74. Диалог настройки отображения переменных в окне **Frame**

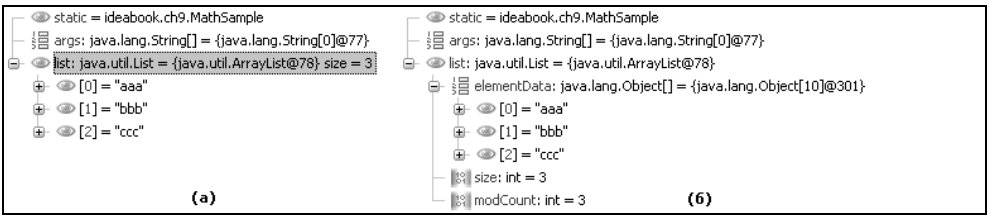


Рис. 9.75. Варианты представления коллекций: *а* — альтернативный; *б* — обычный

- **Show static final fields** (Показывать статические поля, имеющие модификатор `final`);
- **Show synthetic fields** (Показывать синтетические поля) — показывать поля, сгенерированные компилятором;
- **Hide null array elements** (Не отображать элементы массива со значением `null`);
- **Show 'toString()'** (Отображать в виде строкового представления с помощью метода '`toString()`') — набор опций, указывающих способ отображения классов в виде строки (**Types to be shown as 'toString()'**);
- **All** (Все классы);
- **None** (Ни один из классов);

- **Custom** (Выборочно) — данная опция позволяет настроить список классов, которые будут отображаться в виде своего строкового представления. Классы можно задавать как с полного имени (с помощью кнопки **Add** (Добавить)), так и в виде образца (с помощью кнопки **Add pattern** (Добавить образец)), в котором символ '*' позволяет задать произвольный класс или пакет;
- **Show object ID and type when in 'toString()' mode** (Отображать идентификатор и тип объекта, для которого используется режим отображения в виде строки).

9.4.4.1. Дополнительные возможности представления информации

Для переменных целочисленных типов имеется возможность отображать их значения в шестнадцатеричном режиме (рис. 9.76).

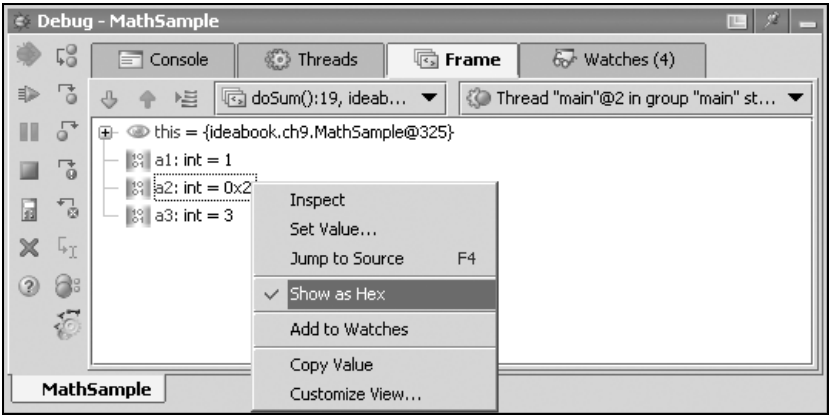


Рис. 9.76. Вывод значения целочисленной переменной в шестнадцатеричном режиме

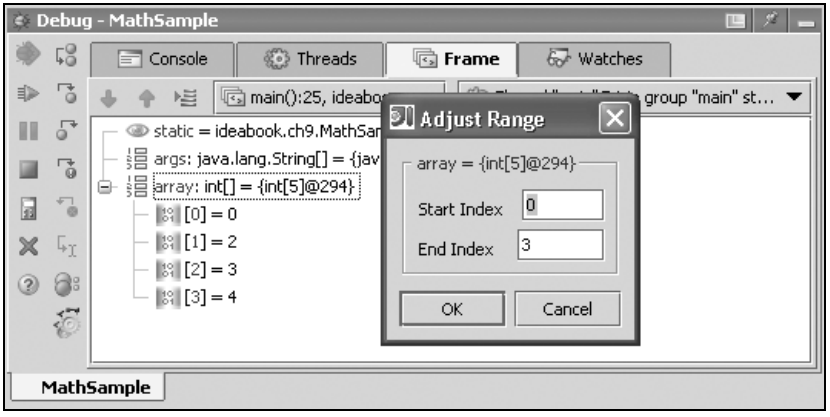


Рис. 9.77. Границы отображения элементов массивов

Для массивов имеется возможность задать границы отображения их элементов (рис. 9.77).

Для переменных-массивов можно ввести начальный индекс (**Start Index**) и конечный индекс (**End Index**) просматриваемых элементов.

9.4.5. *Watches (Наблюдения)*

Если при отладке программы требуется постоянно контролировать значения каких-либо переменных, для удобства можно перенести их в закладку **Watches** (Наблюдения) инструментального окна **Debug** (Отладка) (рис. 9.78).

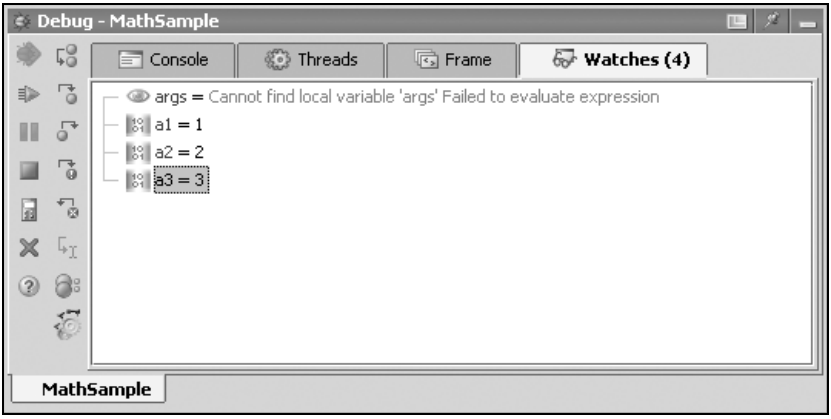


Рис. 9.78. Закладка **Watches** инструментального окна **Debug**

На данной закладке работают все те же самые правила отображения информации, что и на закладке **Frame** (Окно стека).

В список наблюдаемых значений можно добавлять не только отдельные переменные, но и целые выражения.

Для списка наблюдаемых значений работает контекстное меню, которое повторяет контекстное меню списка переменных из закладки **Frame** (Окно стека), но также имеет и четыре дополнительных пункта:

- **New Watch** (Новое выражение для наблюдения) (<Insert>);
- **Edit Watch** (Редактировать наблюдаемое выражение) (<F2>);
- **Remove Watch** (Удалить наблюдаемое выражение) (<Delete>);
- **Remove All Watches** (Удалить все наблюдаемые выражения).

Выражения, входящие в список наблюдаемых, остаются в списке до тех пор, пока их не удалят. Даже если в текущем месте выполнения значение выражения вычислить невозможно, выражение все равно будет находиться в списке (например, выражение `args` в первой позиции списка на рис. 9.78).

9.4.6. Вычисление выражений

Во время отладки имеется возможность вычислять значения произвольных выражений, использующих как значения переменных из контекста программы, так и совершенно произвольных.

Для вычисления выражений предназначена команда **Evaluate Expression** (Вычислить выражение) (<Alt>+<F8>) (рис. 9.79).

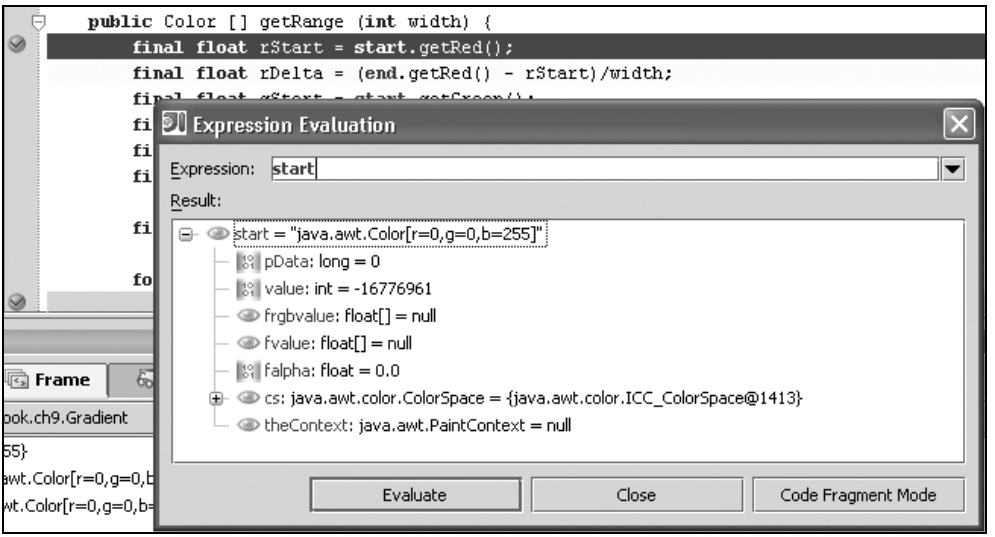


Рис. 9.79. Вычисление выражений

Данная команда открывает диалог, в котором можно ввести любое корректное с точки зрения Java выражение (**Expression**). Чтобы вычислить выражение, нужно нажать кнопку **Evaluate** (Вычислить), и в поле **Result** (Результат) появится результат выражения.

В данном диалоге работают все стандартные команды редактирования (*см. раздел 6.1*).

Диалог имеет два режима работы:

- **Expression Mode** (Режим выражения) — в данном режиме вводится одно выражение, которое затем вычисляется (см. рис. 9.79);
- **Code Fragment Mode** (Режим фрагмента кода) — в данном режиме можно ввести целый фрагмент Java-кода, который затем будет вычислен (рис. 9.80).

В режиме фрагмента кода можно использовать вызовы любых методов, доступных из текущей области видимости. Например, представленный на рис. 9.81.

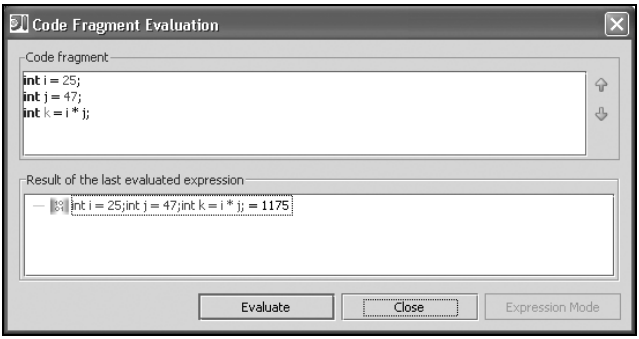


Рис. 9.80. Вычисление выражений: режим фрагмента кода

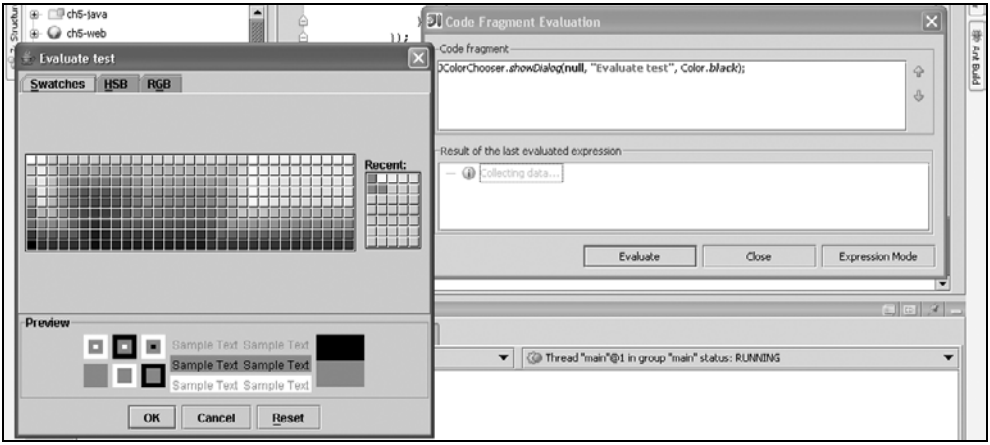


Рис. 9.81. Вычисление фрагмента кода
с использованием вызова метода библиотеки Swing

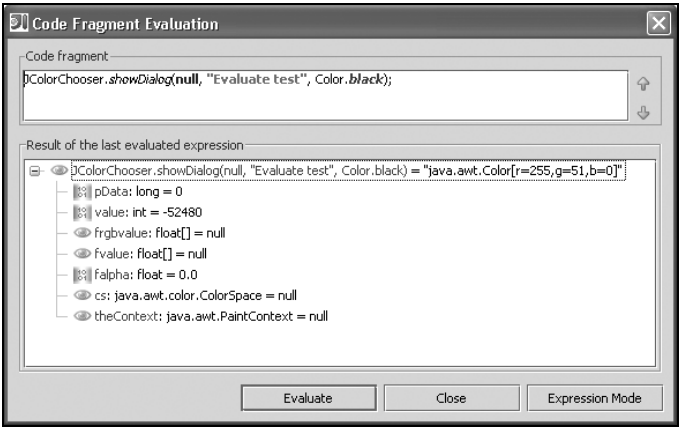


Рис. 9.82. Результат вычисления значения фрагмента кода

После подтверждения диалога результат, который возвратит метод `JColorChooser.showDialog()`, будет отображен на панели просмотра результата (рис. 9.82).

Стрелки в правом верхнем углу диалога предназначены для перебора введенных значений для вычисления.

9.4.7. Функция HotSwap

Если в ходе отладки были внесены некоторые изменения в код программы, в IntelliJ IDEA имеется возможность перезагрузить эти изменения без перезапуска отладчика, эта функция называется *HotSwap* — "горячая замена".

Данная функция чрезвычайно удобна при отладке больших проектов, когда до перехода к коду, в котором установлена точка останова, требуется выполнить некоторый набор действий. Либо когда проект очень долго стартует. В любом случае, "горячая замена" помогает экономить время.

Для выполнения "горячей замены" необходимо выполнить команду **Reload Changes** (Перезагрузить изменения) из меню **Run** (Запуск). Данная команда выполнит сборку измененных классов, а затем перезагрузит их (рис. 9.83).

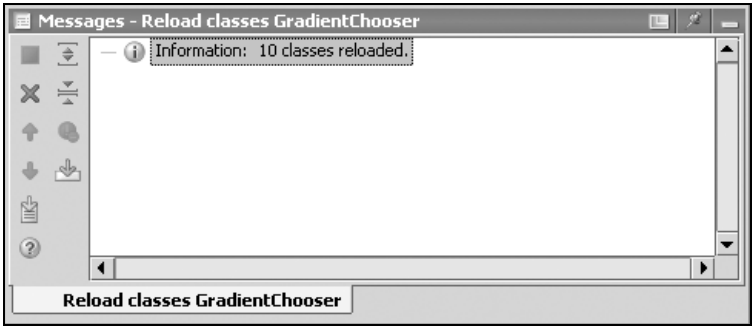
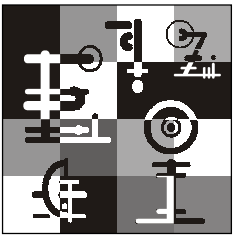


Рис. 9.83. Результат выполнения команды **Reload Changes**

Если в процессе отладки модифицировать некоторые классы, а затем выполнить сборку проекта (или компиляцию отдельных классов), IntelliJ IDEA автоматически предложит перезагрузить изменившиеся классы.

Внимание

Функция "горячей" перезагрузки классов доступна только при отладке приложений, работающих под JDK 1.4 и выше.



Глава 10

J2EE

IntelliJ IDEA обладает встроенными возможностями по созданию и поддержке многих J2EE-сущностей, к числу которых относятся:

- JSP;
- сервлеты;
- различные типы EJB.

С помощью IntelliJ IDEA можно производить запуск и отладку J2EE-приложений, редактировать их конфигурацию, создавать компоненты и управлять ими.

В данной главе мы также расскажем о возможностях IntelliJ IDEA по работе с XML, поскольку XML повсеместно используется для различных дескрипторов J2EE-сущностей.

10.1. Работа с XML

Формат XML поддерживается в IntelliJ IDEA на нескольких уровнях. Во-первых, это поддержка в редакторе, который не только осуществляет подсветку синтаксиса XML, но также позволяет производить автоматическое завершение кода и выводить информацию об атрибутах XML-тега. Во-вторых, IntelliJ IDEA поддерживает типизацию XML с помощью DTD (Document Type Definition) и XMLSchema. Если для некоторого XML указан его тип, IntelliJ IDEA сможет производить для него автоматическую проверку, а также предоставлять некоторые дополнительные возможности по завершению кода.

10.1.1. Редактирование XML

Редактор IntelliJ IDEA имеет много возможностей при редактировании файла в XML-формате.

- Подсветка синтаксиса — в соответствии с глобальными настройками цветов и шрифтов (*см. раздел 4.1.6.4*) в редакторе подсвечиваются различные составляющие XML-файла — теги, атрибуты, их значения, комментарии, данные (рис. 10.1).

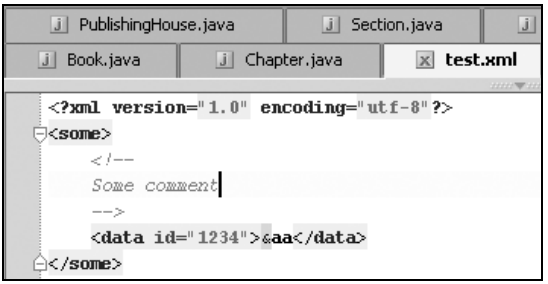


Рис. 10.1. Подсветка синтаксиса в XML

- Автоматическое завершение кода — при редактировании XML работают функции завершения кода, например, подстановка закрывающего тега, если был создан открывающий. Также работает функция простого завершения кода (**Code | Complete | Basic** (Код | Завершение | Простое)) (<Ctrl>+<Space>) (*см. раздел 6.1.1.1*) для имен тегов и их атрибутов (рис. 10.2).

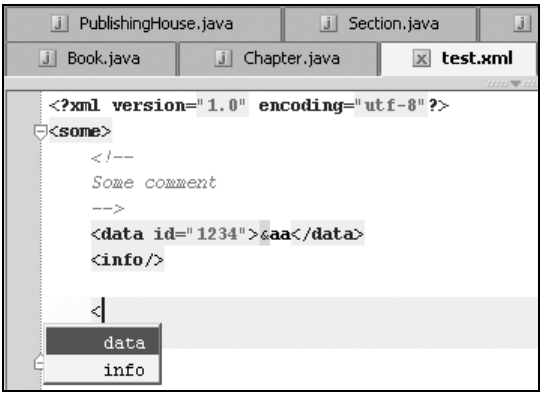


Рис. 10.2. Простое завершение кода

- Проверка ошибок — автоматическая проверка XML-документа на ошибки вроде незакрытых тегов. Для проверки ошибок в XML служит специальная команда **Validate** (Проверить), доступная в контекстном меню XML-файла.

Примечание

При редактировании XML-файла в инструментальном окне **Structure** (Структура) (*см. раздел 3.4.8.8*) отображается структура XML-документа (рис. 10.3).

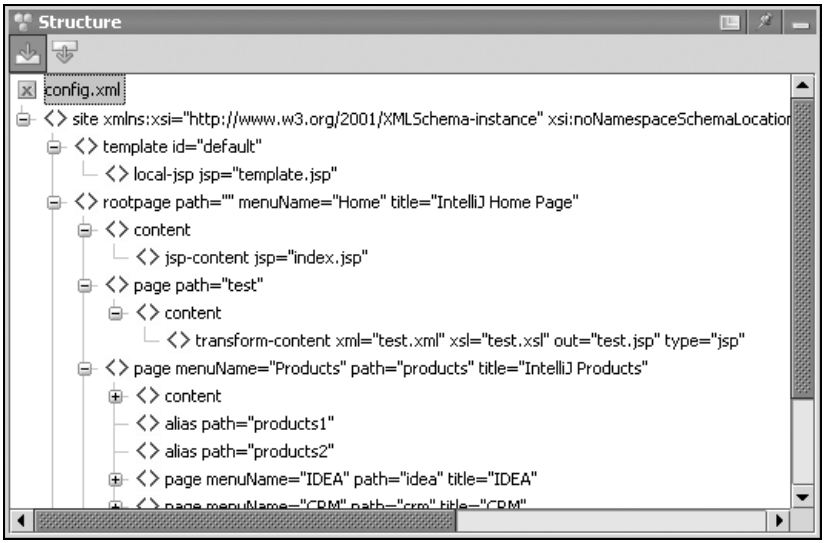


Рис. 10.3. Структура XML-документа в инструментальном окне **Structure**

10.1.2. Поддержка типизации

IntelliJ IDEA поддерживает DTD и XMLSchema для XML-документов. Используя информацию из определения типа документа IntelliJ IDEA может более точно выполнять завершение кода, а также выполнять поиск ошибок в XML-документе и отображать информацию об атрибутах (рис. 10.4).

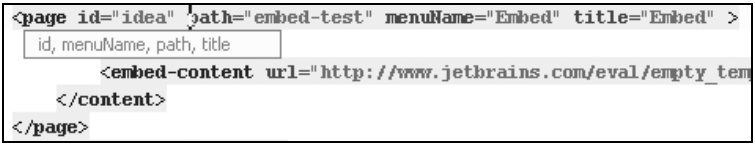


Рис. 10.4. Информация об атрибутах

Из DTD IntelliJ IDEA извлекает информацию о взаимном расположении тегов и их атрибутах.

Для XML-документа, у которого нет ни DTD, ни схемы, IntelliJ IDEA все равно запоминает взаимное расположение тегов и атрибутов и на основе этой информации позволяет выполнять автоматическое завершение кода (рис. 10.5).

Для такого XML-документа IntelliJ IDEA может сгенерировать DTD на основе информации, почерпнутой из самого документа. Для этого можно использовать команду **Generate DTD from XML file** (Создать DTD для XML-файла) из меню **Tools** (Инструменты).

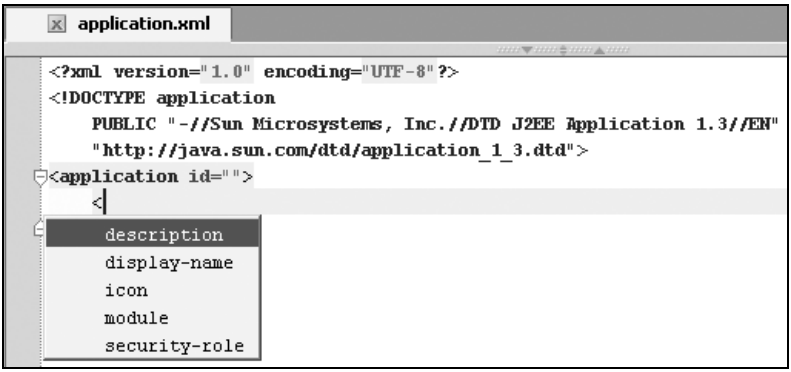


Рис. 10.5. Доступные теги из DTD

10.2. Серверы приложений

IntelliJ IDEA поддерживает три типа серверов приложений:

- Tomcat — JSP-контейнер от Apache/Jakarta;
- WebLogic — сервер приложений компании Bea;
- Generic — абстрактный сервер приложений, заданный набором библиотек. Для запуска и отладки приложений под таким сервером требуется, чтобы он был совместим с JSR-45.

Серверы приложений настраиваются с помощью диалога настройки серверов приложений (*см. раздел 4.1.20*).

Информация о серверах приложений используется IntelliJ IDEA для конфигурации J2EE-модулей в проекте (Web-модулей, EJB-модулей и модулей J2EE-приложений).

10.3. Работа с EJB

EJB в IntelliJ IDEA поддерживается на уровне EJB-модулей (*см. раздел 5.2.3.3*). Каждый EJB-модуль имеет как минимум один дескриптор ejb-jar.xml. Если EJB-модуль использует сервер приложений WebLogic, для него может потребоваться настройка двух дополнительных дескрипторов:

- WebLogic EJB persistence deployment descriptor (weblogic-cmp-rdbms-jar.xml);
- WebLogic EJB Module Deployment Descriptor (weblogic-ejb-jar.xml).

Когда в проекте имеется хотя бы один EJB-модуль, становится доступной закладка **J2EE** инструментального окна **Project** (Проект) (рис. 10.6).

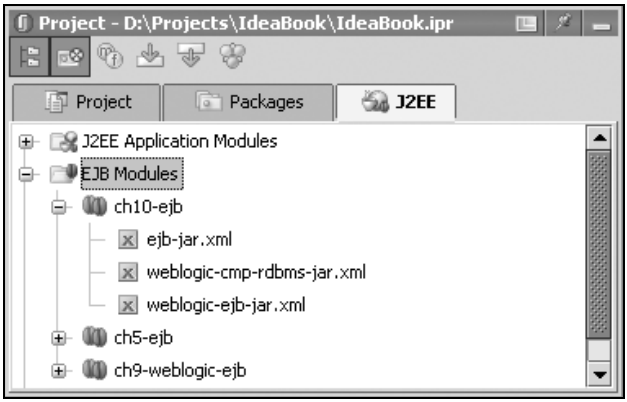


Рис. 10.6. Перечень EJB-модулей в инструментальном окне **Project**

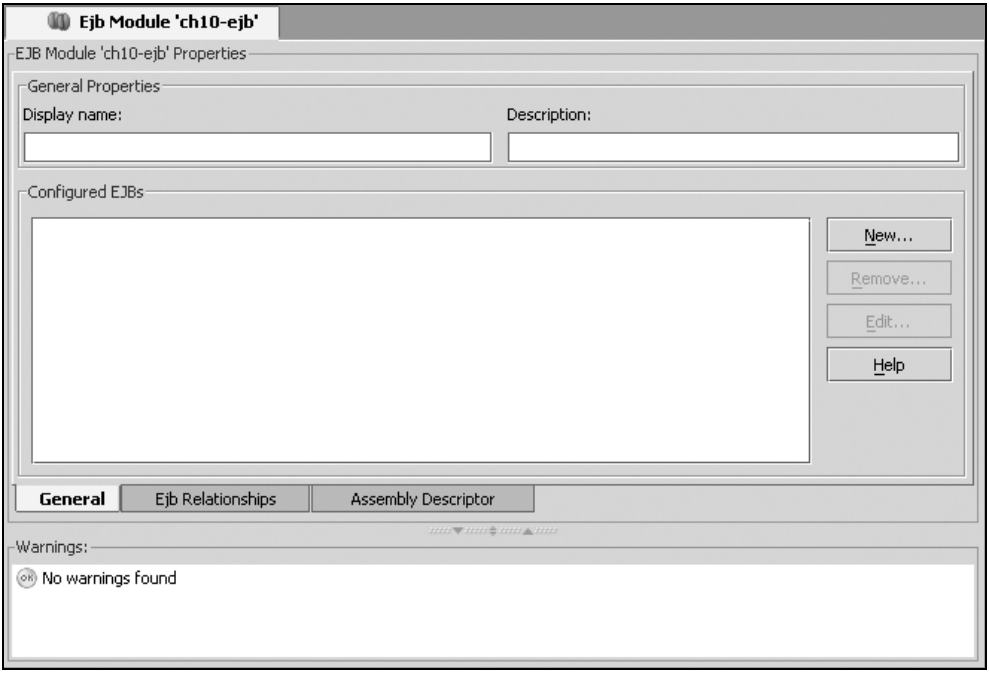


Рис. 10.7. Редактор EJB

Если дважды щелкнуть левой кнопкой мыши на каком-либо EJB-модуле или вызвать команду **Edit** (Редактировать), откроется окно редактора для данного EJB-модуля (рис. 10.7).

Данный редактор, по сути, отражает структуру дескриптора EJB (ejb-jar.xml).

Окно редактора имеет три закладки, в которых располагаются все параметры и настройки EJB:

- **General** (Общие настройки):
  - **Display name** (Имя для отображения EJB-архива);
  - **Description** (Описание);
  - **Configured EJBs** (Сконфигурированные EJB) — список доступных EJB, которые можно создавать (кнопка **New**), удалять (кнопка **Remove**) и редактировать (кнопка **Edit**);
- **Ejb Relationships** (Информация о связях между EJB) — данная закладка содержит список связей между EJB. Связи можно добавлять (кнопка **New**) и удалять (кнопка **Remove**) (*см. раздел 10.3.2*);
- **Assembly Descriptor** (Дескриптор сборки) — данная закладка содержит следующие настройки:
  - **Security Roles** (Роли безопасности) — роли можно добавлять и удалять;
  - **Transaction Attributes** (Атрибуты транзакции) — в данном списке представлены EJB, а также их методы, для которых можно установить одно из следующих значений атрибута транзакции:
    - Not Supported;
    - Required;
    - Supports;
    - RequiresNew;
    - Mandatory;
    - Never;
  - **Method Permissions** (Права доступа методов) — в данном списке можно установить, какие методы не будут проверяться (**Unchecked**), а какие будут исключены (**Excluded**). Если у вас заданы роли безопасности (**Security Roles**), в данном списке прав доступа можно будет также включить поддержку того или иного метода для каждой из ролей безопасности.

В нижней части окна расположено поле **Warnings** (Замечания), в котором выводится информация о проблемах в EJB.

10.3.1. Создание EJB

EJB можно создавать как с помощью кнопки **New** (Создать) в редакторе EJB, так и с помощью контекстного меню EJB-модуля.

Есть четыре EJB, которые умеет создавать IntelliJ IDEA:

- BMP Entity Bean;
- CMP Entity Bean;

- Session Bean;
- Message Bean.

При создании EJB создается не только соответствующий класс, но также и запись в EJB-дескрипторе.

Для создания классов и дескрипторов EJB используются шаблоны файлов (*см. раздел 4.1.14.4*).

10.3.1.1. BMP Entity Bean

BMP Entity Bean создается с помощью диалога, представленного на рис. 10.8.

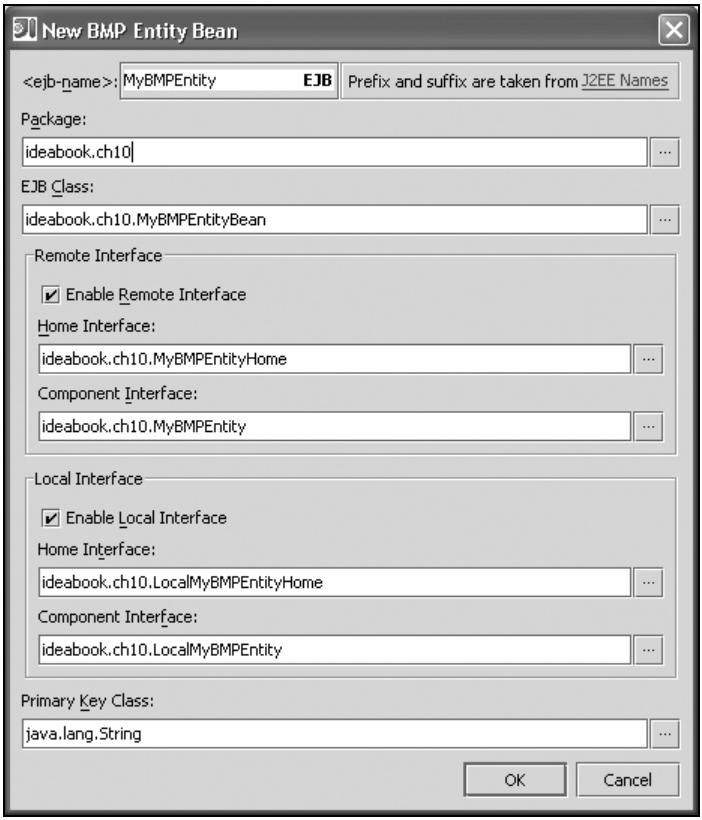


Рис. 10.8. Создание BMP Entity Bean

В данном диалоге можно указать следующие параметры:

- **<ejb-name>** (Имя EJB);
- **Package** (Пакет);
- **EJB Class** (Имя EJB-класса);

- **Remote Interface** (Удаленный интерфейс) — параметры удаленного интерфейса:
  - **Enable Remote Interface** (Включить удаленный интерфейс);
  - **Home Interface** — имя класса для Home Interface;
  - **Component Interface** — имя класса для Component Interface;
- **Local Interface** (Локальный интерфейс) — параметры локального интерфейса:
  - **Enable Local Interface** (Включить локальный интерфейс);
  - **Home Interface** — имя класса для Home Interface;
  - **Component Interface** — имя класса для Component Interface;
- **Primary Key Class** (Класс первичного ключа).

Примечание

IntelliJ IDEA версии 4.5 имеет ошибку, которая проявляется при создании EJB. Если директория для исходных файлов модуля имеет префикс пакета (*см. раздел 5.3.1*), то при создании EJB могут возникнуть проблемы с указанием правильного пути пакета.

После подтверждения диалога дескриптор EJB примет вид, представленный в листинге 10.1.

Листинг 10.1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar
  PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
         'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
    <description>EJB Sample</description>
    <display-name>ch10ejb</display-name>
    <enterprise-beans>
        <entity>
            <description>My EJB description</description>
            <display-name>MyEJB</display-name>
            <ejb-name>MyBMPEntityEJB</ejb-name>
            <home>ideabook.ch10.MyBMPEntityHome</home>
            <remote>ideabook.ch10.MyBMPEntity</remote>
            <local-home>ideabook.ch10.LocalMyBMPEntityHome</local-home>
            <local>ideabook.ch10.LocalMyBMPEntity</local>
```

```
            <ejb-class>ideabook.ch10.MyBMPEntityBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.String</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>MyBMPEntityEJB</abstract-schema-name>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
    </assembly-descriptor>
</ejb-jar>
```

После того как EJB создан, он появится в списке EJB-модуля, а также для него будет открыт отдельный редактор (рис. 10.9).

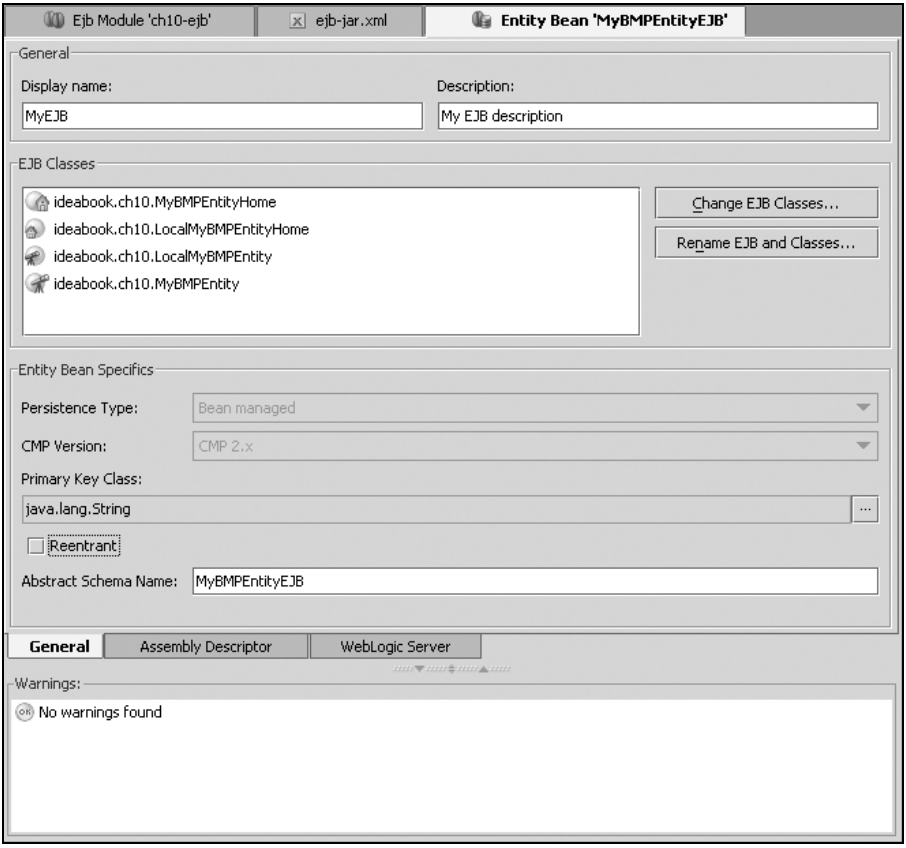


Рис. 10.9. Редактирование BMP EJB

В редакторе BMP EJB имеются три закладки:

- **General** (Общие свойства):
  - **Display name** (Имя для отображения);
  - **Description** (Описание);
  - **EJB Classes** (Классы EJB) — список классов EJB. Классы можно редактировать и переименовывать с помощью диалога, представленного на рис. 10.8. Для изменения EJB-классов можно использовать кнопку **Change EJB Classes** (Изменить классы EJB), а для переименования EJB — кнопку **Rename EJB and Classes** (Переименовать EJB и классы);
- **Entity Bean Specifics** (Дополнительные опции Entity Bean):
  - **Persistence Type**;
  - **CMP Version** (Версия CMP);
  - **Primary Key Class** (Класс для первичного ключа);
  - **Reentrant**;
  - **Abstract Schema Name** (Имя абстрактной схемы);

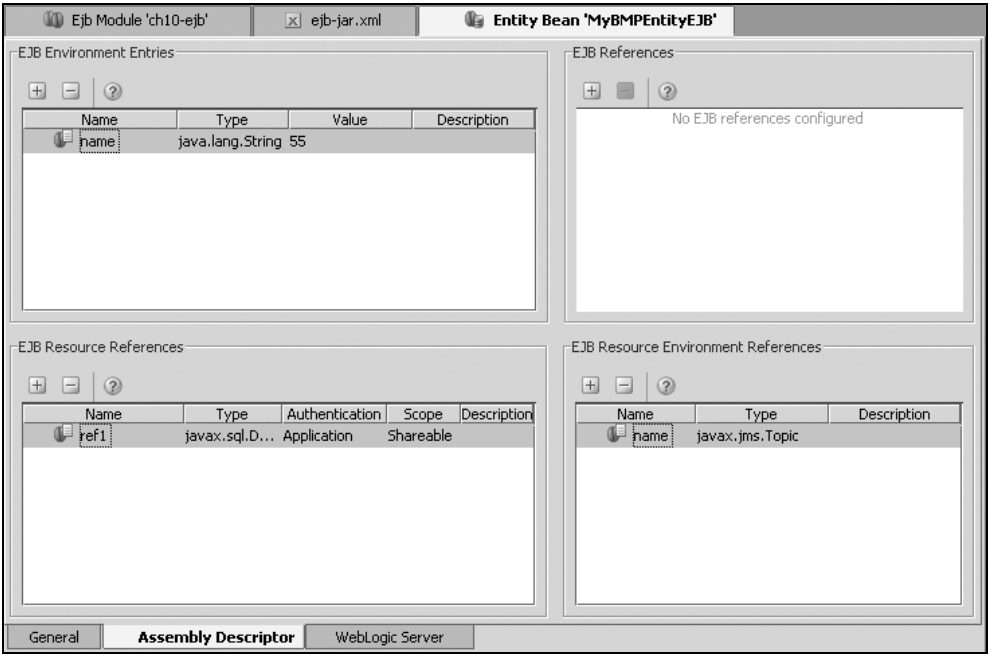


Рис. 10.10. Параметры дескриптора сборки EJB

- **Assembly Descriptor** (Дескриптор сборки) — на данной закладке расположены четыре панели с настройками (рис. 10.10):
  - **EJB Environment Entries** (Записи EJB-окружения) — параметры можно добавлять и удалять. Для записей можно задать:
  - **Name** (Имя);
  - **Type** (Тип) — один из типов `java.lang.String`, `java.lang.Character`, `java.lang.Integer`, `java.lang.Boolean`, `java.lang.Double`, `java.lang.Float`;
  - **Value** (Значение);
  - **Description** (Описание);

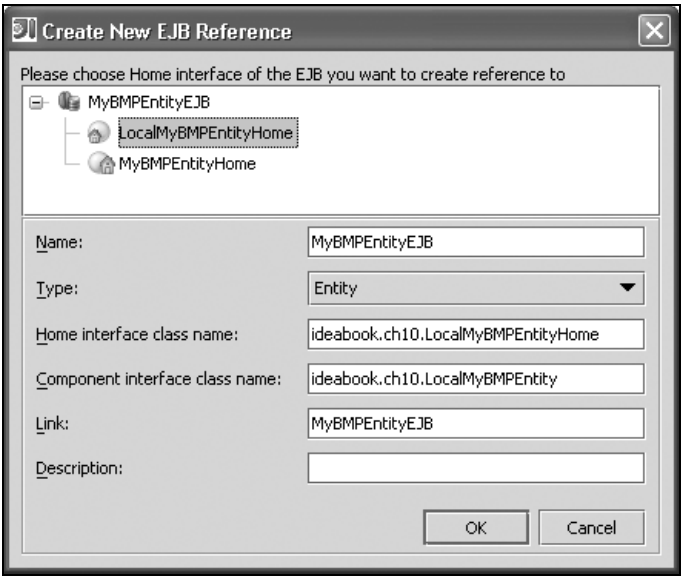


Рис. 10.11. Добавление ссылки на EJB

- **EJB References** (Ссылки на EJB) — ссылки можно добавлять и удалять. Для добавления ссылок используется диалог, представленный на рис. 10.11. С помощью данного диалога можно выбрать:
  - ◊ **Home Interface**, с которым будет установлена связь;
  - ◊ **Name** (Имя);
  - ◊ **Type** (Тип) — одно из двух значений: **Entity** или **Session**;
  - ◊ **Home interface class name** (Имя класса **Home Interface**);
  - ◊ **Component interface class name** (Имя класса **Component interface**);
  - ◊ **Link** (Связь);
  - ◊ **Description** (Описание);

- **EJB Resource References** (Ссылки на ресурсы EJB) — ссылки можно добавлять и удалять. Для ссылок на ресурсы можно задать:
  - ◊ **Name** (Имя);
  - ◊ **Type** (Тип) — один из следующих типов: `javax.sql.DataSource`, `java.net.URL`, `javax.mail.Session`, `javax.jms.QueueConnectionFactory`, `javax.jms.TopicConnectionFactory`, `javax.jms.XAQueueConnectionFactory`, `javax.jms.XATopicConnectionFactory`;
  - ◊ **Authentication** (Аутентификация) — тип аутентификации: **Container** (Аутентификация контейнером) или **Application** (Аутентификация приложением);
  - ◊ **Scope** (Область видимости) — **Shareable** (Разделяемый) или **Unshareable** (Неразделяемый);
  - ◊ **Description** (Описание);
- **EJB Resource Environment References** (Ссылки на ресурсы EJB из окружения). Для ссылок на ресурсы EJB из окружения можно задать:
  - ◊ **Name** (Имя);
  - ◊ **Type** (Тип) — одно из следующих значений: `javax.jms.Queue` или `javax.jms.Topic`;
  - ◊ **Description** (Описание);

❐ **WebLogic Server** (Настройки для сервера WebLogic) — данная закладка доступна, только если ваш модуль использует сервер приложений WebLogic. На данной закладке расположены следующие параметры:

- **JNDI Name** (Имя JNDI);
- **Local JNDI Name** (Локальное имя JNDI).

10.3.1.2. CMP Entity Bean

Для создания CMP Entity Bean используется диалог, практически идентичный представленному на рис. 10.8. Единственным отличием является возможность выбрать значение параметра **CMP Version** (Версия CMP) — 1.x или 2.x.

Редактор CMP Entity Bean также похож на редактор BMP Entity Bean, однако имеются и отличия (рис. 10.12).

На закладке **General** (Общие) имеется дополнительный список полей (**CMP Fields**), которые можно добавлять (кнопка **Add CMP Field**), удалять (кнопка **Remove CMP Field**) и редактировать (кнопка **Edit CMP Field**).

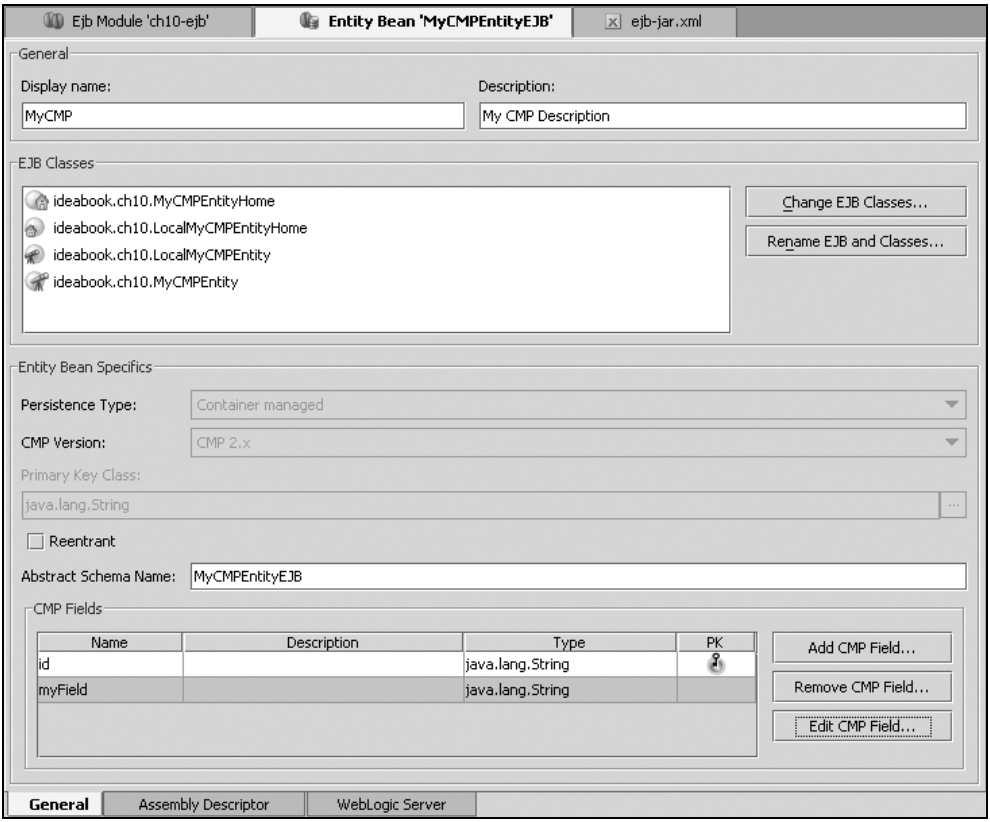


Рис. 10.12. Редактирование CMP Entity Bean

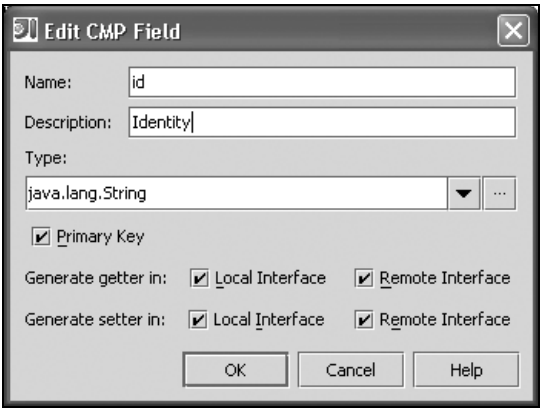


Рис. 10.13. Редактирование поля CMP

Добавление и редактирование полей CMP производится с помощью диалога, представленного на рис. 10.13.

В данном диалоге есть следующие параметры и опции:

- **Name** (Имя поля);
- **Description** (Описание поля);
- **Type** (Тип поля) — можно выбрать предопределенный тип поля, либо задать свой собственный;
- **Primary Key** (Первичный ключ) — опция, делающая поле первичным ключом;
- **Generate getter in** (Создать метод `get` в):
  - **Local Interface** (Локальный интерфейс);
  - **Remote Interface** (Удаленный интерфейс);
- **Generate setter in** (Создать метод `set` в):
  - **Local Interface** (Локальный интерфейс);
  - **Remote Interface** (Удаленный интерфейс).

Закладка **Assembly Descriptor** (Дескриптор сборки) аналогична одноименной закладке BMP Entity Bean (*см. раздел 10.3.1.1*).

Закладка **WebLogic Server** (данная закладка доступна, если ваш EJB-модуль сконфигурирован использовать сервер приложений WebLogic) содержит дополнительные параметры и опции (рис. 10.14).

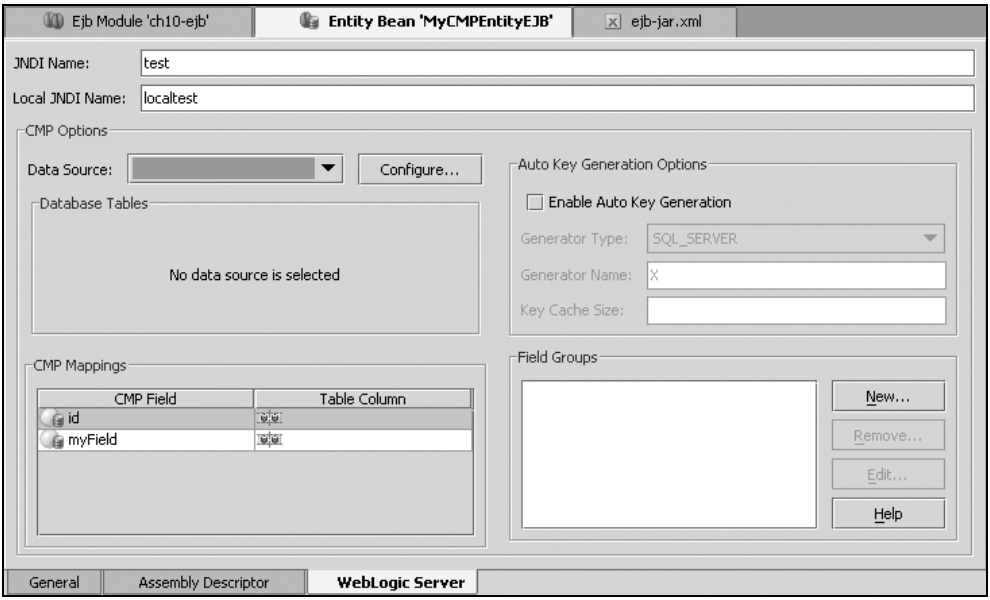


Рис. 10.14. CMP Entity Bean сконфигурирован использовать WebLogic Server

Рис. 10.15. Источники данных

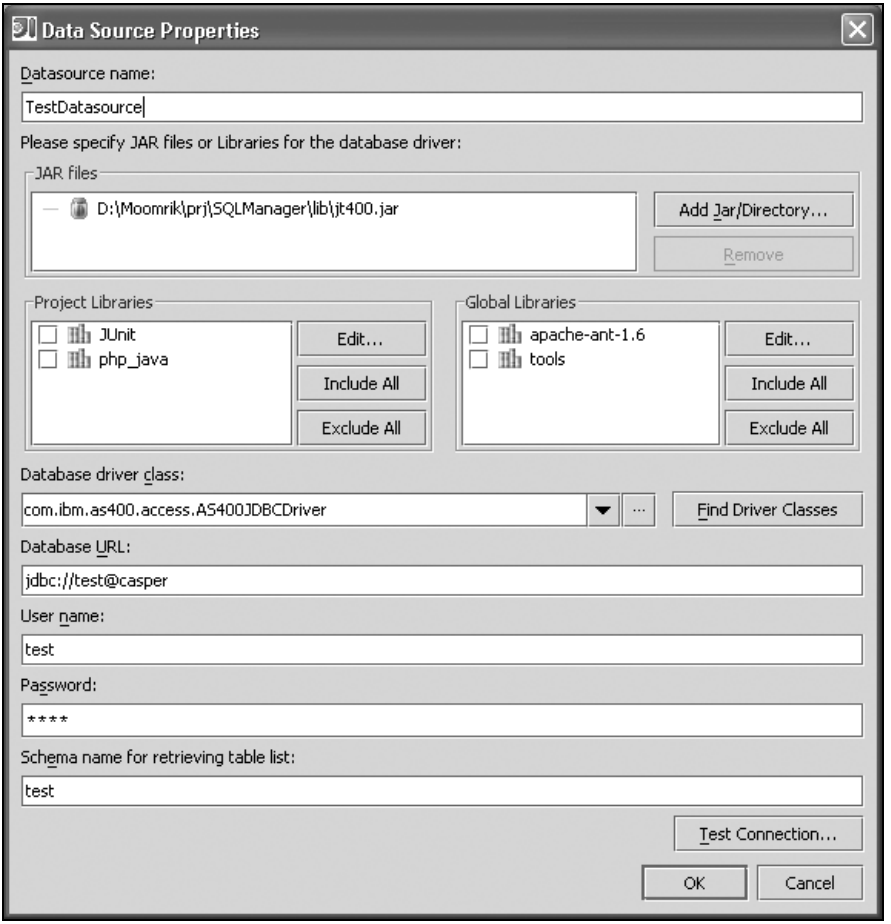


Рис. 10.16. Свойства источника данных

На данной закладке расположены следующие опции и параметры:

- **Data Source** (Источник данных) — источник данных можно выбрать из списка доступных, либо сконфигурировать новые с помощью кнопки **Configure** (Конфигурировать). Данная кнопка открывает диалог, представленный на рис. 10.15.

  С помощью данного диалога можно добавлять, удалять и настраивать свойства источников данных. Создание и редактирование свойств источников данных производится с помощью диалога, представленного на рис. 10.16.

  С помощью данного диалога можно настроить следующие свойства источника данных:

  - **Datasource name** (Имя источника данных);
  - **Please specify JAR files or Libraries for the database driver** (JAR-файлы или библиотеки с драйвером базы данных) — имеется возможность выбрать либо JAR-файлы, либо библиотеки проекта или глобальные;
  - **Database driver class** (Класс драйвера базы данных). С помощью кнопки **Find Driver Classes** (Найти классы драйверов) можно автоматически вывести список доступных классов;
  - **Database URL** (URL базы данных);
  - **User name** (Имя пользователя);
  - **Password** (Пароль);
  - **Schema name for retrieving table list** (Имя схемы для получения списка таблиц);

  С помощью кнопки **Test Connection** (Проверить соединение) можно проверить соединение с базой данных;

- **Database Tables** (Таблицы в базе данных). Таблицы можно добавлять (кнопка **New** (Создать)) и удалять (кнопка **Remove** (Удалить));
- **CMP Mappings** — соответствие между полями CMP и полями в таблице из базы данных;
- **Auto Key Generation Options** (Параметры автоматической генерации ключей):
  - **Enable Auto Key Generation** (Включить автоматическую генерацию ключей);
  - **Generator Type** (Тип генерации):
    - ◊ `ORACLE`;
    - ◊ `SQL_SERVER`;
    - ◊ `NAMED_SEQUENCE_TABLE`;
  - **Generator Name** (Имя генератора);
  - **Key Cache Size** (Размер кэша ключей);

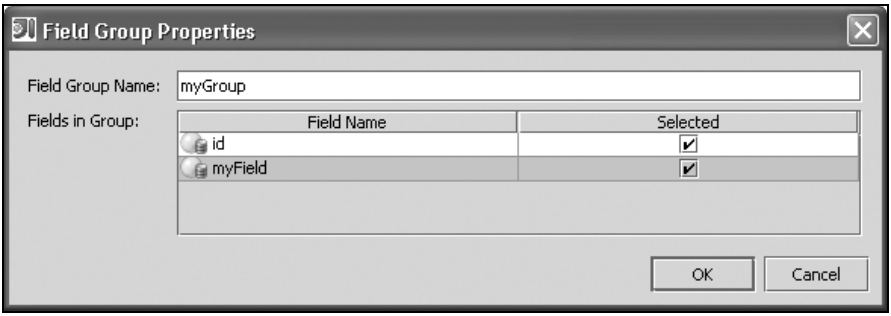


Рис. 10.17. Свойства группы полей

- **Field Groups** (Группы полей). Группы полей можно добавлять (кнопка **New** (Создать)), удалять (кнопка **Remove** (Удалить)) и редактировать (кнопка **Edit** (Редактировать)). Для добавления и редактирования используется диалог, представленный на рис. 10.17.

  В диалоге свойств группы полей можно указать название группы (**Field Group Name** (Название группы полей)), а также выбрать поля, которые будут входить в группу (**Fields in Group** (Поля в группе)).

10.3.1.3. Session Bean

Создание Session Bean практически ничем не отличается от создания BMP Entity Bean (см. рис. 10.8).

Редактор свойств Session Bean представлен на рис. 10.18.

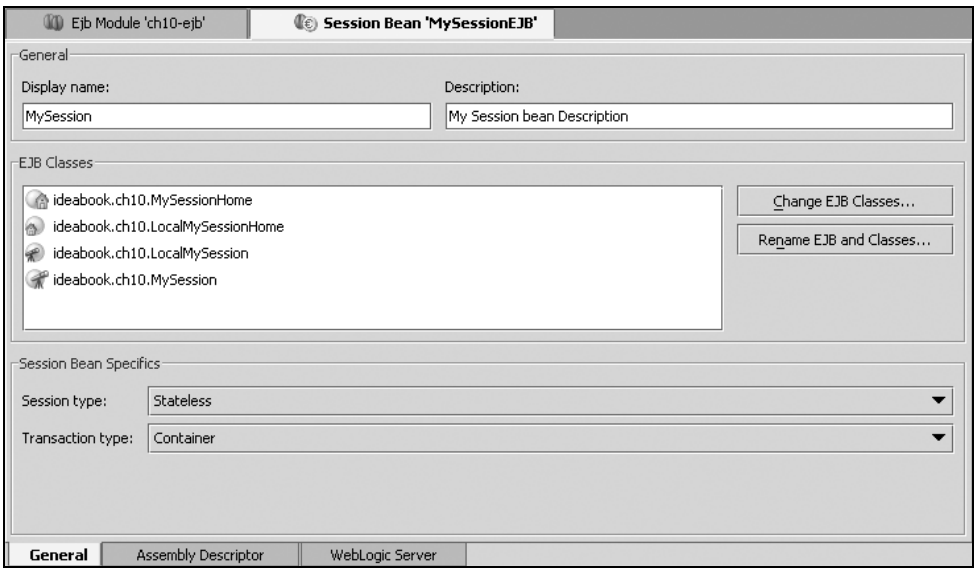


Рис. 10.18. Редактор Session Bean

Кроме имени (**Display name**), описания (**Description**) и EJB-классов (**EJB Classes**), для Session Bean имеется возможность задать два параметра:

- **Session type** (Тип сессии):
  - **Stateless**;
  - **Stateful**;
- **Transaction type** (Тип транзакции):
  - **Bean**;
  - **Container**.

Все остальные параметры Session Bean идентичны BMP Entity Bean (*см. раздел 10.3.1.1*).

10.3.1.4. Message Bean

Создание Message Bean осуществляется с помощью диалога, представленного на рис. 10.19.

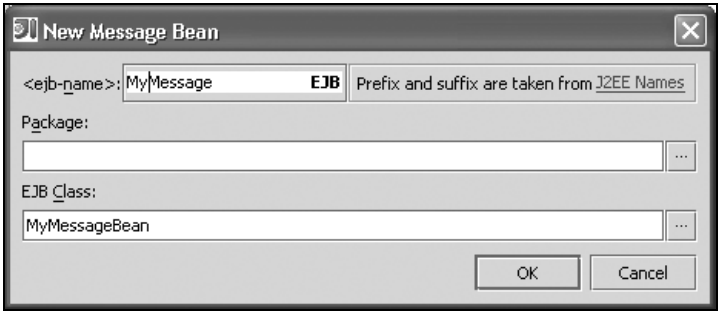


Рис. 10.19. Создание Message Bean

В данном диалоге можно задать следующие параметры:

- **<ejb-name>** — имя EJB;
- **Package** (Пакет);
- **EJB Class** (Имя EJB-класса).

Редактор Message Bean представлен на рис. 10.20.

Кроме имени EJB (**Display name**), описания (**Description**) и классов (**EJB Classes**), с помощью редактора имеется возможность задать следующие параметры:

- **Transaction type** (Тип транзакции):
  - **Bean**;
  - **Container**;

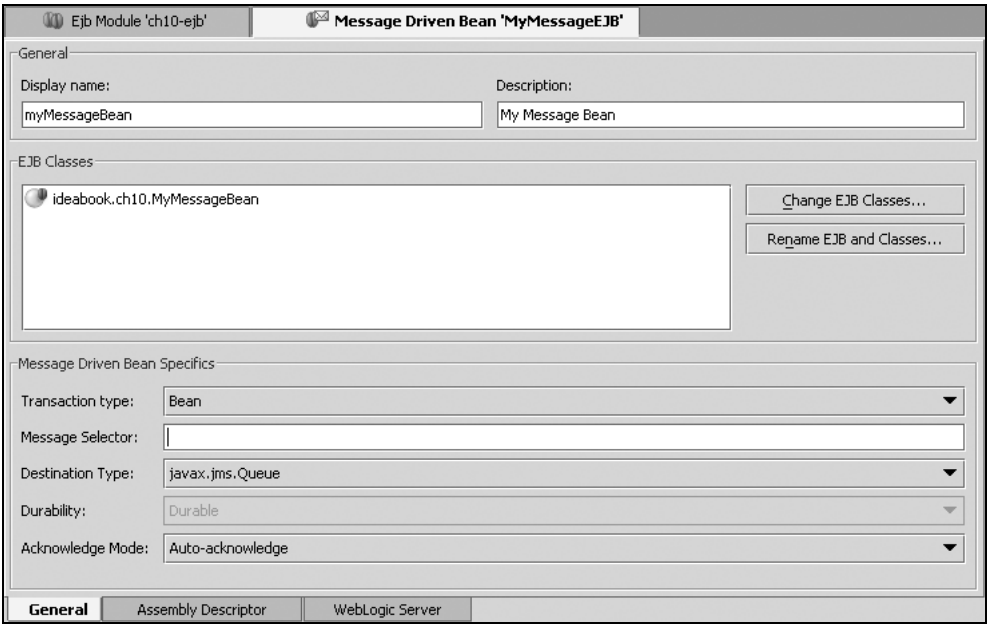


Рис. 10.20. Редактор Message Bean

- **Message Selector** (Селектор сообщения);
- **Destination Type** (Тип назначения):
  - `javax.jms.Queue`;
  - `javax.jms.Topic`;
- **Durability** (Длительность);
- **Acknowledge Mode** (Режим допуска):
  - **Auto-acknowledge**;
  - **Dups-ok-acknowledge**.

Параметры на закладках **Assembly Descriptor** (Дескриптор сборки) и **WebLogic Server** аналогичны BMP Entity Bean (*см. раздел 10.3.1.1*).

10.3.2. Установка связей между EJB

Создание связей между EJB производится с помощью диалога, представленного на рис. 10.21.

С помощью данного диалога можно настроить следующие параметры:

- **Relationship Name** (Название связи);
- **Description** (Описание);

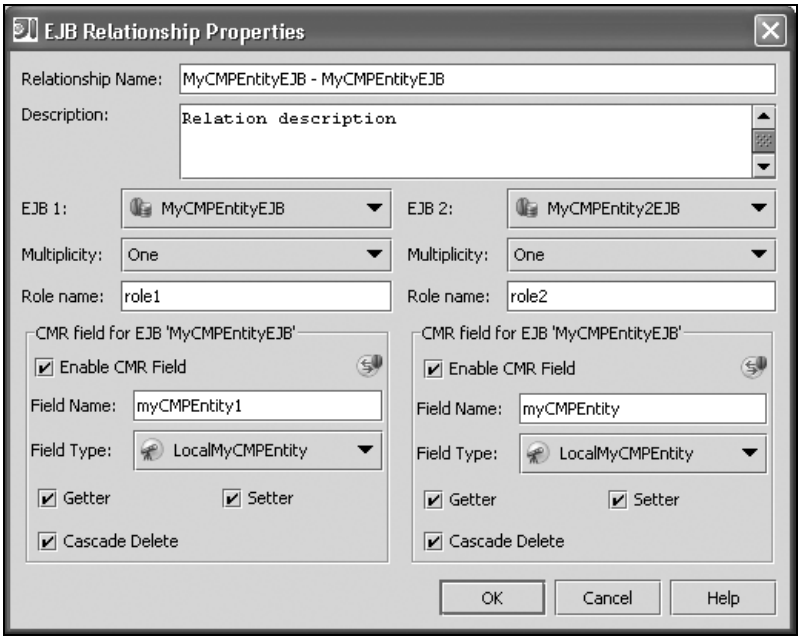


Рис. 10.21. Создание связи между EJB

- **EJB 1** и **EJB 2** — параметры для двух связываемых EJB:
  - **EJB** (Название EJB);
  - **Multiplicity** (Тип связи) — имеется возможность задать четыре типа связей между EJB:
    - ◊ **One-One** (Один-к-одному);
    - ◊ **One-Many** (Один-ко-многим);
    - ◊ **Many-One** (Многие-к-одному);
    - ◊ **Many-Many** (Многие-ко-многим);
  - **Role name** (Имя роли);
  - **CMR field for EJB** (Поле CMR для EJB):
    - ◊ **Enable CMR Field** (Включить поле CMR);
    - ◊ **Field Name** (Имя поля);
    - ◊ **Field Type** (Тип поля);
    - ◊ **Getter** (Метод `get`);
    - ◊ **Setter** (Метод `set`);
    - ◊ **Cascade delete** (Каскадное удаление).

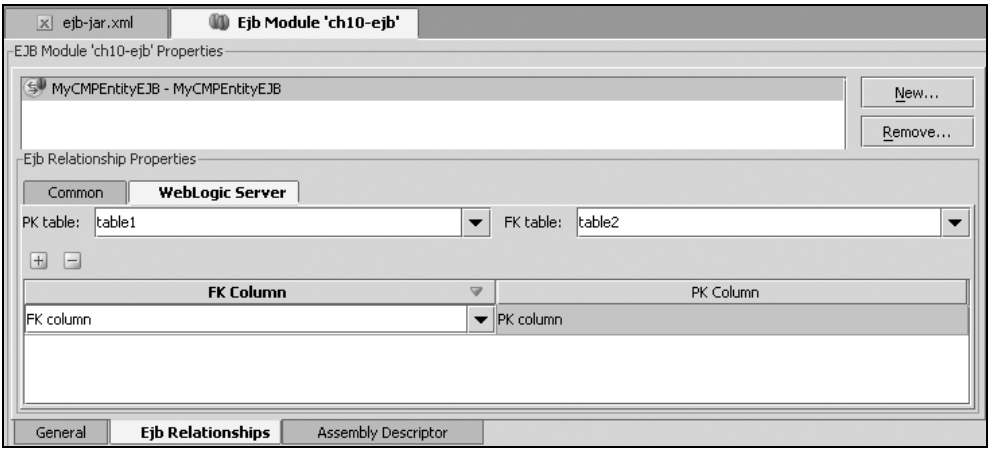


Рис. 10.22. Дополнительные настройки связей EJB для WebLogic

Если для модуля используется сервер приложений WebLogic, на закладке **Ejb Relationships** (Отношения EJB) появится дополнительная закладка с настройками WebLogic (рис. 10.22).

С помощью параметров на данной закладке можно задать соответствие между колонками в таблицах.

10.4. Web-приложения

Web-приложения поддерживаются в IntelliJ IDEA на уровне Web-модулей (*см. раздел 5.2.3.2*). В контексте Web-приложений имеется поддержка JSP, сервлетов, а также редактора компонентов Web-приложения.

Когда в проекте IntelliJ IDEA имеется хотя бы один Web-модуль, он будет отображаться в списке доступных Web-модулей на закладке **J2EE** инструментального окна **Project** (Проект) (рис. 10.23).

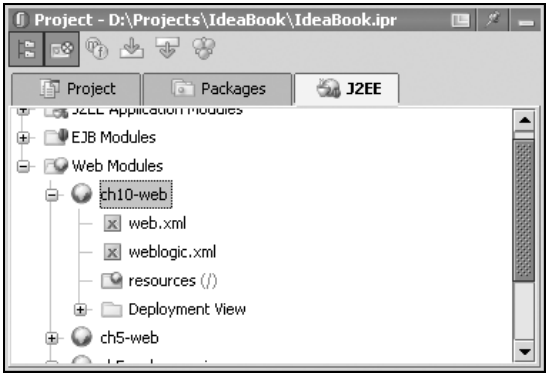


Рис. 10.23. Список доступных Web-модулей

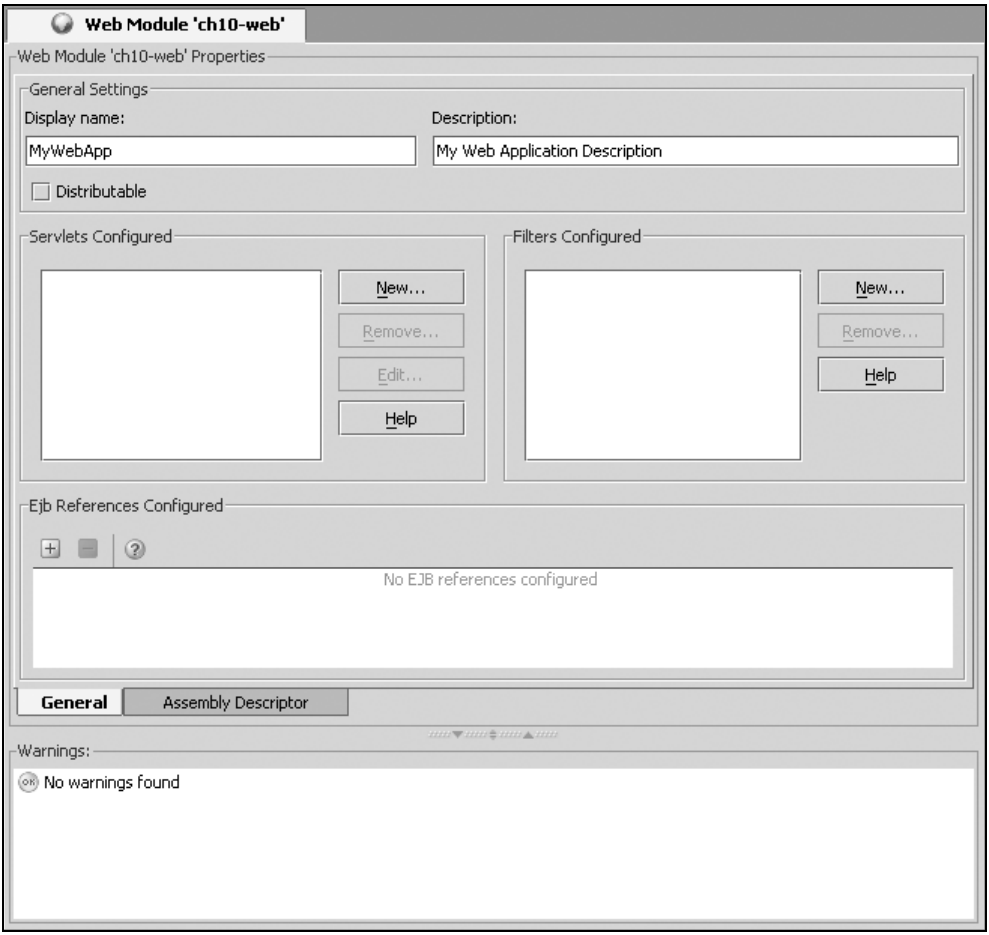


Рис. 10.24. Редактор Web-модуля

Если дважды щелкнуть левой кнопкой мыши на имени Web-модуля, либо вызвать команду **Edit** (Редактировать) из контекстного меню модуля, откроется редактор Web-модуля (рис. 10.24).

Опции и параметры, представленные в данном редакторе отражают структуру дескриптора Web-модуля web.xml.

Параметры и опции в редакторе Web-модуля расположены на двух закладках:

- **General** (Общие настройки):
  - **Display name** (Имя для отображения);
  - **Description** (Описание);

- **Distributable** (Распределенное) — опция, позволяющая разрешить выполнять Web-приложение на нескольких серверах в распределенном окружении (например, на нескольких кластерах);
- **Servlets Configured** (Сервлеты) — список сервлетов, которые можно добавлять (кнопка **New** (Создать)), удалять (кнопка **Remove** (Удалить)) и редактировать (кнопка **Edit** (Редактировать));
- **Filters Configured** (Фильтры) — список фильтров, которые можно добавлять (кнопка **New** (Создать)), удалять (кнопка **Remove** (Удалять)) и редактировать (кнопка **Edit** (Редактировать));
- **Ejb References Configured** (Ссылки на EJB) — список ссылок на EJB. Ссылки можно добавлять, удалять и редактировать с помощью соответствующих команд на панели инструментов, а также с помощью команд из контекстного меню. Для добавления ссылок используется диалог, представленный на рис. 10.11;

❒ **Assembly Descriptor** (Дескриптор сборки) — с помощью параметров данной закладки можно настроить (рис. 10.25):

- **Security Roles** (Роли безопасности);
- **Context Parameters** (Параметры контекста);
- **Servlet Mappings** (Карта соответствия сервлетов);
- **Filter Mappings** (Карта соответствия фильтров).

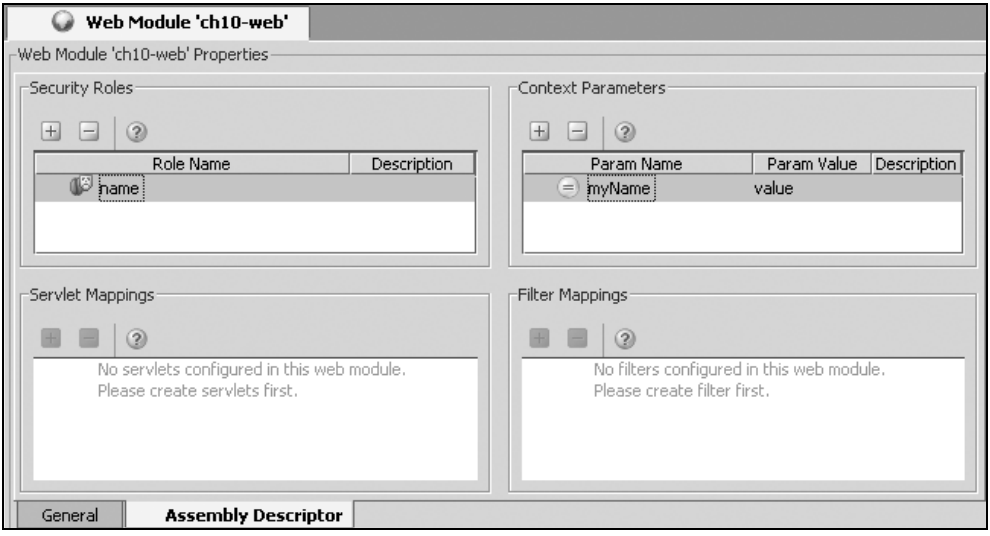


Рис. 10.25. Дескриптор сборки Web-приложения

10.4.1. Сервлеты

IntelliJ IDEA поддерживает автоматизированное создание сервлетов, редактирование их свойств, а также настройку соответствий (mapping) для сервлетов.

10.4.1.1. Создание сервлетов

Для создания сервлета необходимо либо нажать кнопку **New** (Создать) на панели **Servlets Configured** (Сервлеты) редактора Web-модуля, либо выполнить команду **New** (Создать) контекстного меню Web-модуля.

Сервлет создается с помощью диалога, представленного на рис. 10.26.

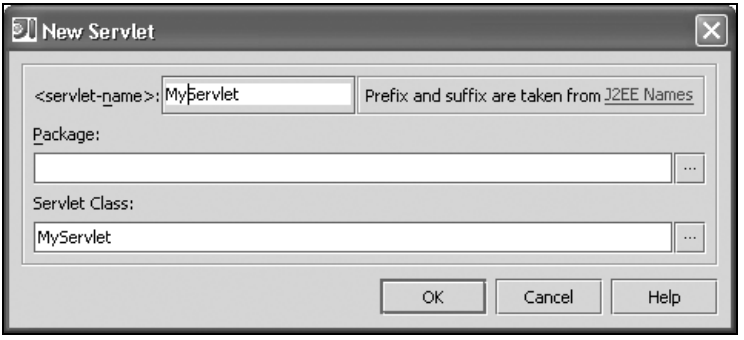


Рис. 10.26. Создание сервлета

При создании сервлета необходимо указать:

- **<servlet-name>** (Имя сервлета);
- **Package** (Пакет);
- **Servlet Class** (Класс сервлета).

После создания сервлета в дескрипторе Web-приложения (web.xml) создается соответствующая запись. Также создается класс, соответствующий сервлету. Редактирование сервлета производится с помощью редактора сервлета (рис. 10.27).

В редакторе сервлета можно указать следующие параметры:

- **Display name** (Имя для отображения);
- **Description** (Описание);
- **Servlet Initialization Params** (Параметры инициализации сервлета). Параметры можно добавлять, удалять, менять их имена (**Param Name**), значения (**Param Value**) и описания (**Description**);
- **Load on startup** (Загружать при старте Web-приложения);
- **Load order** (Порядок загрузки) — порядковый номер сервлета при загрузке.

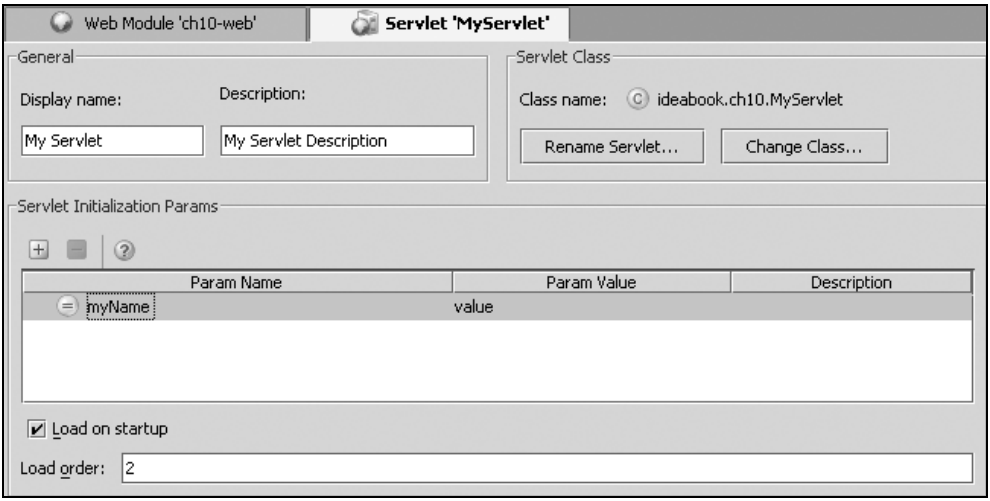


Рис. 10.27. Редактор сервлета

Сервлет можно переименовать с помощью кнопки **Rename Servlet** (Переименовать сервлет), а также изменить класс сервлета с помощью кнопки **Change Class** (Изменить класс).

10.4.1.2. Привязка сервлетов

Настройка привязки сервлетов осуществляется с помощью панели **Servlet Mappings** (Привязка сервлетов) на закладке **Assembly Descriptor** (Дескриптор сборки) редактора Web-приложения (рис. 10.28).

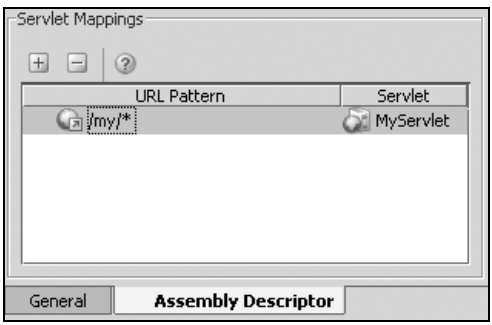


Рис. 10.28. Привязка сервлетов

Для привязки сервлета к некоторому URL необходимо добавить новую запись в список, а затем указать необходимые значения для полей **URL Pattern** (Образец URL) и **Servlet** (Сервлет).

10.4.2. Фильтры

IntelliJ IDEA поддерживает автоматизированное создание фильтров, а также настройку соответствий (mapping) для фильтров.

10.4.2.1. Создание фильтров

Создание фильтра выполняется аналогично созданию сервлета. Для задания параметров создания фильтра используется диалог, представленный на рис. 10.29.

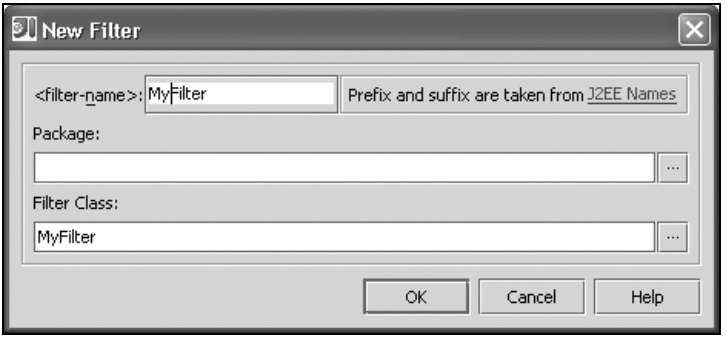


Рис. 10.29. Создание фильтра

10.4.2.2. Привязка фильтров

Настройка привязки фильтров осуществляется с помощью панели **Filter Mappings** (Привязка фильтров) на закладке **Assembly Descriptor** (Дескриптор сборки) редактора Web-приложения (рис. 10.30).

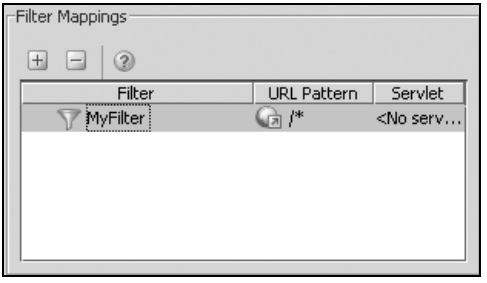


Рис. 10.30. Привязка фильтров

Для привязки фильтра к некоторому URL необходимо добавить новую запись в список, а затем указать необходимые значения для полей **Filter** (Фильтр), **URL Pattern** (Образец URL) и **Servlet** (Сервлет) — последнее поле можно не заполнять, если это не требуется.

10.4.3. JSP

IntelliJ IDEA поддерживает JSP на уровне редактора, позволяя работать с кодом JSP как с обычным Java-кодом. Дополнительно к этому имеется встроенная поддержка TagLib расширений.

Для создания JSP-файла можно воспользоваться командой **Edit | New** (Редактирование | Создать), вызванной для директории, соответствующей директории с ресурсами Web-модуля (рис. 10.31).

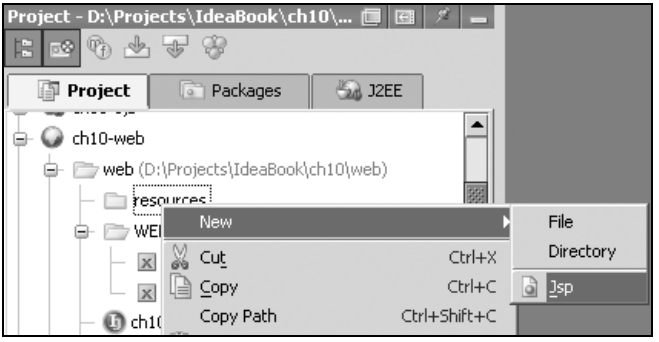


Рис. 10.31. Создание JSP-файла

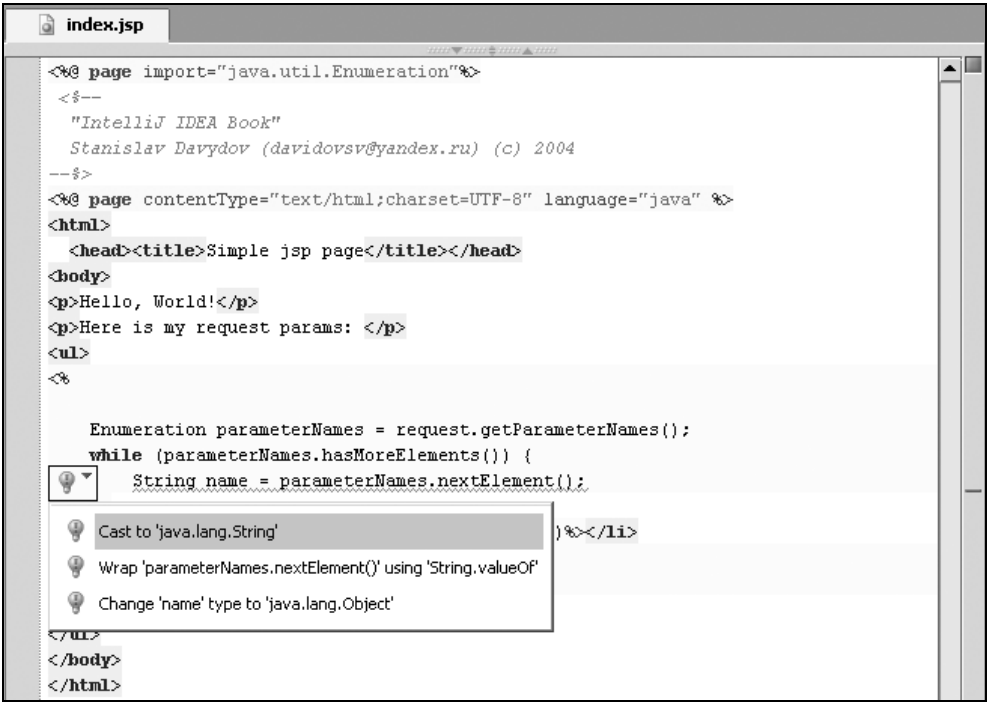


Рис. 10.32. Редактор JSP

Для создания JSP-файла используется соответствующий шаблон из шаблонов файлов IntelliJ IDEA (*см. раздел 4.1.14*).

В редакторе JSP-файла работают большинство команд редактирования, завершения кода, генерации кода как для скриптовых вставок, так и для специальных JSP-тегов. Также выполняется подсветка синтаксиса и ошибок, работают "подсказки" (рис. 10.32).

При написании тегов JSP имеется возможность автоматического завершения названий тегов и их атрибутов (рис. 10.33).

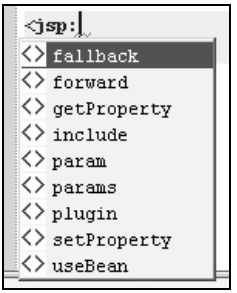


Рис. 10.33. Завершение имени JSP-тега

Можно сказать, что редактор JSP в IntelliJ IDEA совмещает в себе возможности редактора Java-кода (*см. раздел 6.1*) и редактора XML (*см. раздел 10.1.1*).

Если ваше Web-приложение использует библиотеку расширений тегов (TagLib), IntelliJ IDEA сможет применять расширенные теги при завершении кода (рис. 10.34).

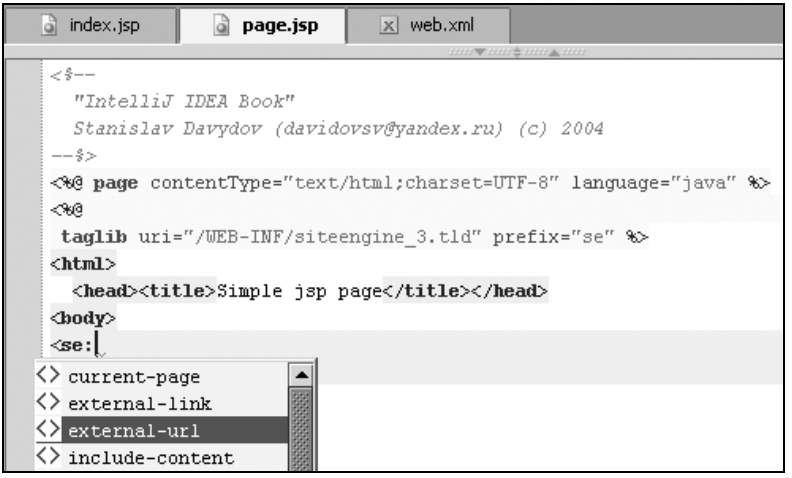


Рис. 10.34. Завершение названия расширенного тега

10.5. Приложения J2EE

IntelliJ IDEA поддерживает J2EE-приложения для сервера приложений WebLogic, а также для произвольных JSR45-совместимых серверов. J2EE-приложения поддерживаются на уровне модулей J2EE-приложений (*см. раздел 5.2.3.4*).

Если в проекте имеется хотя бы один модуль J2EE-приложения, он будет отображен в списке доступных модулей J2EE-приложений на закладке **J2EE** инструментального окна **Project** (Проект) (рис. 10.35).

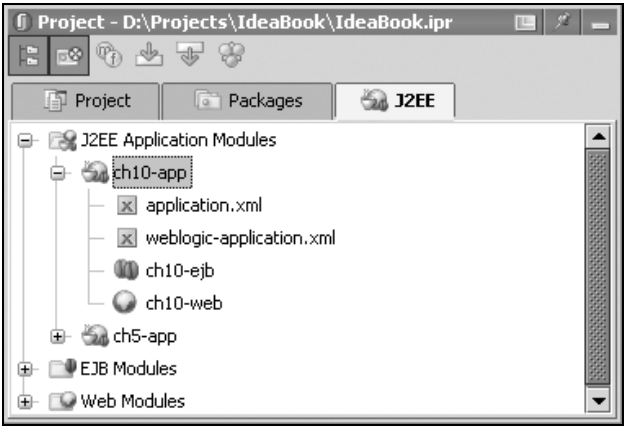


Рис. 10.35. Список модулей J2EE-приложений

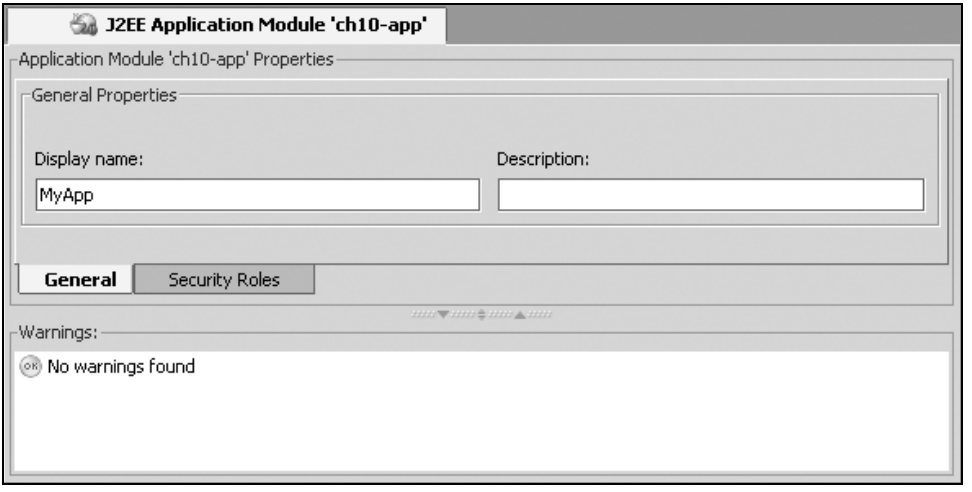


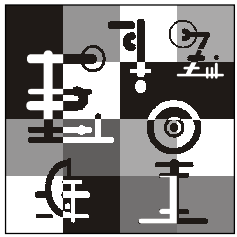
Рис. 10.36. Редактор J2EE-приложения

Если дважды щелкнуть левой кнопкой мыши модуль J2EE-приложения, либо выполнить команду **Edit** (Редактировать) из контекстного меню, будет открыт редактор J2EE-приложения, который, по сути, отражает структуру J2EE-приложения, описанную в дескрипторе J2EE-приложения (application.xml) (рис. 10.36).

В данном редакторе имеются две закладки с параметрами J2EE-приложения:

- **General Properties** (Основные параметры):
  - **Display name** (Имя для отображения);
  - **Description** (Описание);
- **Security Roles** (Роли безопасности) — на данной закладке расположен список ролей безопасности, которые можно добавлять, удалять, менять их название (**Role Name**) и описание (**Description**).

В нижней части редактора J2EE-приложений расположено поле сообщений о замечаниях и ошибках **Warnings** (Замечания).



Глава 11

Поддержка новых возможностей Java 5

Java 2 Platform Standard Edition 5.0 Development Kit (JDK 5.0) — последняя версия пакета разработки на языке Java фирмы Sun. Еще до того, как данный пакет был выпущен, IntelliJ IDEA уже имела ограниченную поддержку его новых возможностей.

Сейчас IntelliJ IDEA поддерживает более широкий круг новых возможностей языка Java версии 5:

- Generics;
- Enhanced for (улучшенный цикл `for` для коллекций);
- Autoboxing/Autounboxing (автоматическая конвертация между простыми типами и объектами);
- Enum (перечислимые типы);
- Varargs (функции с переменным числом аргументов);
- Static import (статические импорты);
- Annotations (аннотации кода).

Прежде чем использовать в коде новые возможности языка Java, необходимо соответствующим образом настроить проект:

1. Включить поддержку возможностей языка Java версии 5 (*см. раздел 4.2.1*).
2. Установить JDK 5.0 и указать IntelliJ IDEA использовать его как JDK проекта. В качестве альтернативы можно установить отдельную поддержку JSR-014 (*см. раздел 3.2.11.6*), однако в данном случае IntelliJ IDEA сможет корректно работать только с параметризованными классами (*см. раздел 11.1*).
3. Если вы используете пакет расширения JSR-014, необходимо дополнительно установить опцию **Use generics-enabled compiler** (Использовать компилятор, поддерживающий параметризованные классы) (*см. раздел 4.2.2*).

Далее мы рассмотрим возможности IntelliJ IDEA при работе со всеми перечисленными нововведениями языка Java.

11.1. Generics

Generics — это способ параметризации Java-классов (листинг 11.1).

Листинг 11.1

```
public class GenericsSample1 {
    public static void main(String[] args) {
        final List <String> argList = Arrays.asList(args);
        for (int i = 0; i < argList.size(); i++) {
            final String arg = argList.get(i);
            System.out.println("arg = " + arg);
        }
    }
}
```

IntelliJ IDEA имеет полную поддержку параметризованных классов при создании Java-кода, включая автоматическое завершение кода, проверку ошибок, подсветку синтаксиса (рис. 11.1), вывод "подсказок", автоматическую проверку инспекциями, вывод информации о коде и т. д. (*см. раздел 6.1*), т. е. работают все функции IntelliJ IDEA, как и для обычного Java-кода.

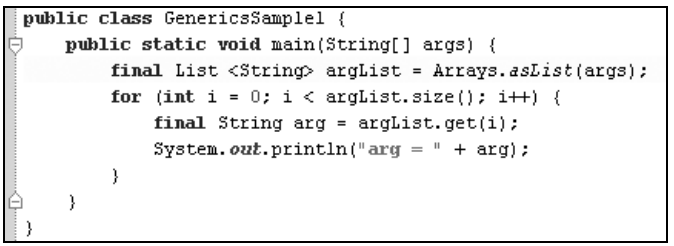


Рис. 11.1. Пример Java-кода, использующего параметризованные классы

Можно взять более сложный код (листинг 11.2). Данный код содержит ошибку, которую IntelliJ IDEA находит и подсвечивает (рис. 11.2).

Листинг 11.2

```
public class GenericsSample2 {
    public static void main(String[] args) {
        List<String> [] stringLists = new List<String>[] {
```

```
            new ArrayList <Boolean> (
                    Arrays.asList(new Boolean [] {true, false, true}))
        };
        for (int i = 0; i < stringLists.length; i++) {
            List<String> stringList = stringLists[i];
            for (int j = 0; j < stringList.size(); j++) {
                String s = stringList.get(j);
                System.out.println("s = " + s);
            }
        }
    }
}
```

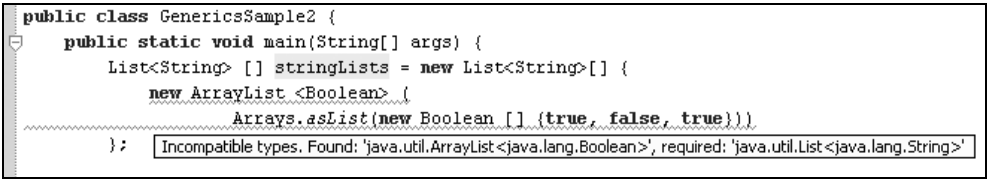


Рис. 11.2. IntelliJ IDEA подсвечивает ошибку в коде, использующем параметризованные классы

11.2. Цикл *for* для коллекций

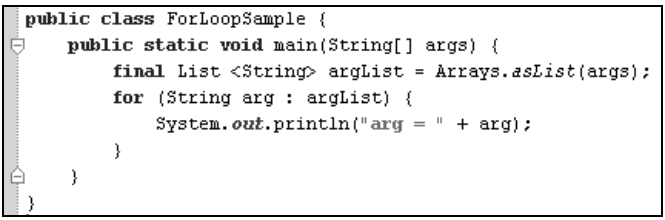
В Java 5 появился новый синтаксис цикла `for` для коллекций и массивов (листинг 11.3).

Листинг 11.3

```
public class ForLoopSample {
    public static void main(String[] args) {
        final List <String> argList = Arrays.asList(args);
        for (String arg : argList) {
            System.out.println("arg = " + arg);
        }
    }
}
```

IntelliJ IDEA поддерживает данную языковую конструкцию, т. е. распознает ее как верную (рис. 11.3). Также в IntelliJ IDEA имеется специальный

"живой шаблон" (*см. раздел 6.3.2*), который может создать код с использованием данной формы цикла `for`.



```
public class ForLoopSample {
    public static void main(String[] args) {
        final List <String> argList = Arrays.asList(args);
        for (String arg : argList) {
            System.out.println("arg = " + arg);
        }
    }
}
```

Рис. 11.3. Новая форма цикла `for`

11.3. Auto(un)boxing

Внятно и коротко перевести данный термин на русский язык не представляется возможным[1], поэтому мы будем упоминать его в английской нотации, записанной русскими буквами:

- Autoboxing ("автобоксинг") — автоматическое преобразование простых типов данных, например `int`, `boolean` (и других), в объекты соответствующих им типов, например `Integer`, `Boolean` (и пр.);
- Autounboxing ("автоанбоксинг") — обратный процесс, когда объекты типов `Integer`, `Boolean` (и других) автоматически преобразуются в простые типы `int`, `boolean` и пр.

Данное преобразование позволяет значительно упростить программные конструкции. Так, например, в листинге 11.2 используется "автобоксинг", при заполнении массива элементов типа `Boolean`.

Пример "автоанбоксинга" представлен в листинге 11.4.

Листинг 11.4

```
public class BoxingSample {
    public static void main(String[] args) {
        // autoboxing
        final Integer [] intArray = {1, 2, Integer.valueOf(33), 21};

        // autounboxing
        int sum = 0;
        for (Integer integer : intArray) {
```

[1] Возможные варианты перевода: "заворачивание" и "разворачивание".

```
            sum += integer;
        }

        System.out.println("sum = " + sum);
    }
}
```

IntelliJ IDEA поддерживает использование "автобоксинг" и "автоанбоксинг" в Java-коде (рис. 11.4).

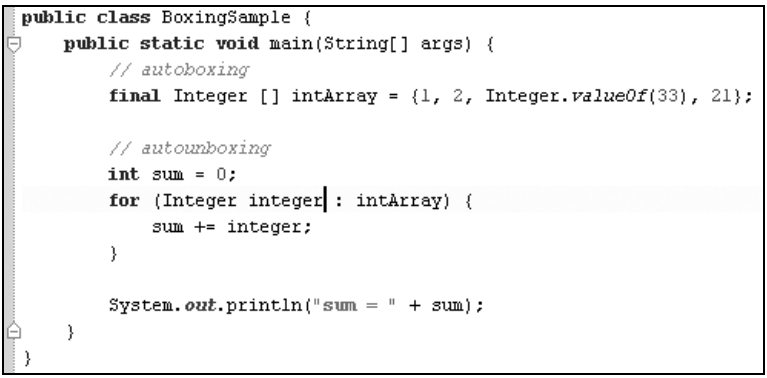


Рис. 11.4. Поддержка auto(un)boxing

11.4. Перечислимый тип *enum*

Перечислимый тип, появившийся в Java 5, позволил значительно сократить код и сделать его более понятным. Раньше при использовании переменных, количество значений которых ограничено (перечислений), приходилось применять один из двух подходов:

1. Создавать несколько констант предопределенного типа, которые использовались в программе в качестве значений перечислимых переменных (листинг 11.5).
2. Создавать специальные классы, которые содержали в себе некий аналог решения из предыдущего пункта, но защищенный с точки зрения типизации (см. листинг 3.1).

Листинг 11.5

```
public class DataKeeper {
    public static final int INDEX_RED = 0;
    public static final int INDEX_GREEN = 1;
```

```
    public static final int INDEX_BLUE = 2;

    public static final int STATE_INC = 0;
    public static final int STATE_DEC = 1;
    public static final int STATE_NO = 2;
}
```

В Java 5 вместо этого можно использовать специальные перечислимые объекты `enum` (листинг 11.6).

Листинг 11.6

```
public class EnumSample {
    enum Index { red, green, blue };
    enum State { inc, dec, none };

    Index index;
    State state;
}
```

В IntelliJ IDEA имеется поддержка перечислимых типов на том же уровне, что и любых других типов (рис. 11.5).

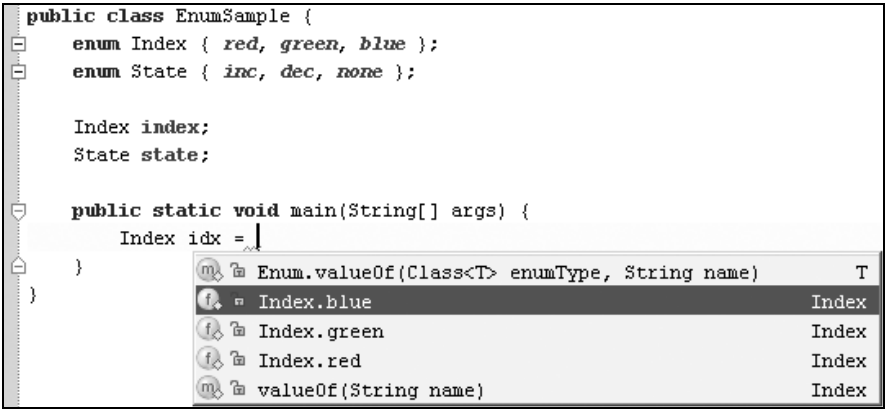


Рис. 11.5. Поддержка перечислимых типов `enum`: создание

Прочие расширенные возможности перечислимых типов также поддерживаются (рис. 11.6).

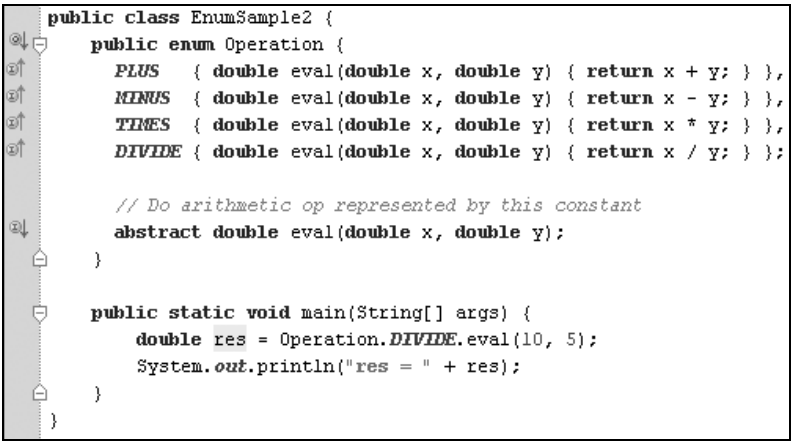


Рис. 11.6. Поддержка перечислимых типов `enum`: результат

11.5. Переменное число параметров метода

В Java 5 добавлена возможность создавать методы с переменным числом параметров (листинг 11.7).

Листинг 11.7

```
public class VarargSample {
    public static void printAll (Object... params) {
        for (int i = 0; i < params.length; i++) {
            Object param = params[i];
            System.out.println("param = " + param);
        }
    }

    public static void main(String[] args) {
        printAll("param1", 22, "param3");
    }
}
```

IntelliJ IDEA поддерживает такие методы на уровне редактирования кода, а также позволяет использовать "живой шаблон" для создания итерации по параметрам (рис. 11.7).

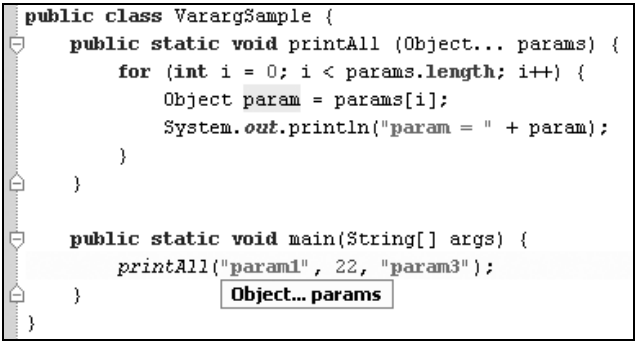


Рис. 11.7. Метод с переменным числом параметров

11.6. Статические импорты

Еще одним новшеством в Java 5 стали статические импорты (static imports) (листинг 11.8).

Листинг 11.8

```
import static java.lang.Math.*;

import javax.swing.*;

public class StaticImportSample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "<html>\u03c0 = " +
                        PI + "</p></html>");
    }
}
```

IntelliJ IDEA поддерживает статические импорты (рис. 11.8).

Для статических импортов, как и для обычных, имеется проверка использования. Если ни одна константа из класса статического импорта не используется, такая строка импорта будет помечена как неиспользуемая.

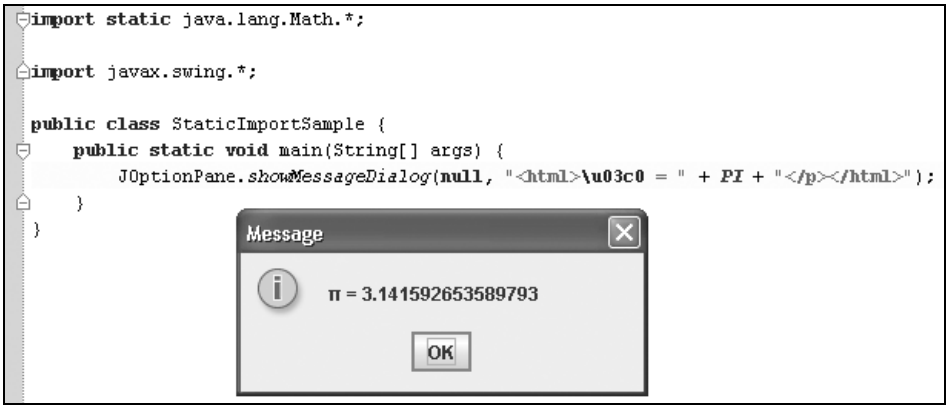


Рис. 11.8. Статический импорт

11.7. Аннотации

Аннотации (annotations) — специальный механизм встраивания дополнительной информации в Java-код. В старых версиях Java (1.4 и ниже) тоже имеется механизм аннотаций, самой распространенной из которых является, наверное, аннотация `@deprecated`, записываемая для методов и классов, которые устарели и которые больше не рекомендуется использовать (листинг 11.9).

Листинг 11.9

```
/**
 * @deprecated This method is deprecated, use method oof() instead
 */
public void foo () {
    // do something
}
public void oof () {
}
```

В Java 5 механизм аннотаций значительно расширен. Для аннотаций определен специальный синтаксис, API для чтения аннотаций и даже специальная утилита для обработки аннотаций `apt`.

Аннотации не меняют логику работы программы, а лишь выполняют вспомогательные функции при обработке программы внешними утилитами. Аннотации еще называют метаданными (metadata).

В листинге 11.10 приведен пример использования аннотаций.

Листинг 11.10

```
public @interface AuthorInfo {
    String name();
    String email();
}

public class AnnotationSample2 {
    @AuthorInfo (
            name = "Stanilsav Davydov",
            email = "davydovsv@yandex.ru"
        )
    public void foo () {
    }
}
```

IntelliJ IDEA поддерживает аннотации на уровне редактирования кода и позволяет просматривать информацию о параметрах аннотации и подсвечивать ошибки в использовании аннотаций (рис. 11.9, 11.10).

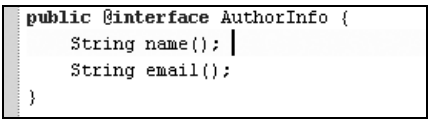


Рис. 11.9. Аннотация

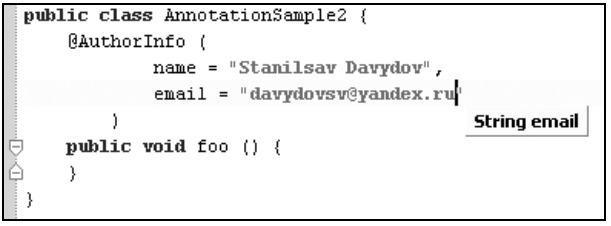
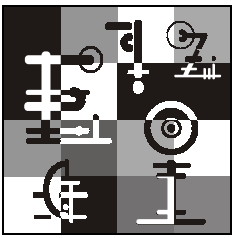


Рис. 11.10. Использование аннотаций

Для создания аннотаций в IntelliJ IDEA имеется специальная команда в контекстном меню **Edit | New** (Редактирование | Создать) (*см. раздел 3.2.2.10*).



Часть III

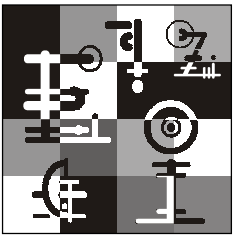
Интеграция

Глава 12. Системы контроля версий

Глава 13. Утилита Ant

Глава 14. Тестирование

Глава 15. Плагины



Глава 12

Системы контроля версий

IntelliJ IDEA имеет встроенную поддержку нескольких систем контроля версий:

- CVS;
- SourceSafe;
- StarTeam.

Также в IntelliJ IDEA имеется встроенная локальная система контроля версий, автоматически отслеживающая все изменения, происходящие с директориями и файлами проекта — Local History (Локальная история).

Рассказ о системах контроля версий и их поддержке в IntelliJ IDEA мы начнем с описания встроенной утилиты сравнения файлов (Diff) (*см. раздел 12.1*). Затем в *разделе 12.2* мы расскажем о локальной системе контроля версий. Оставшуюся часть главы (*разделы 12.3—12.5*) мы посвятим рассмотрению интеграции IntelliJ IDEA с тремя перечисленными системами контроля версий.

12.1. Утилита Diff

IntelliJ IDEA имеет встроенную утилиту сравнения файлов. Чаще всего данная утилита используется при работе с версиями файлов проекта, сохраненными в системе контроля версий. Однако возможности данной утилиты не ограничиваются только системами контроля версий. С ее помощью можно сравнивать любые файлы проекта между собой, а также можно сравнивать части файлов.

12.1.1. Сравнение двух файлов

Чтобы сравнить два файла, необходимо открыть один из них в редакторе, а для второго файла выполнить команду **Compare File with Editor** (Сравнить

файл с редактором). Данная команда присутствует в контекстном меню инструментальных окон **Project** (Проект) (*см. раздел 3.4.8.2*) и **Commander** (Коммандер) (*см. раздел 3.4.8.3*).

12.1.2. Сравнение частей файлов

Чтобы сравнить произвольные части любых файлов, необходимо скопировать в буфер обмена часть текста, которую надо сравнить. Выделить другую часть текста, которую нужно сравнить. Части могут находиться как в одном, так и в разных файлах. Затем для данной выделенной области выполнить команду из контекстного меню **Compare with Clipboard** (Сравнить с буфером обмена).

12.1.3. Представление результатов сравнения

Результаты сравнения представляются с помощью окна утилиты сравнения файлов (рис. 12.1).

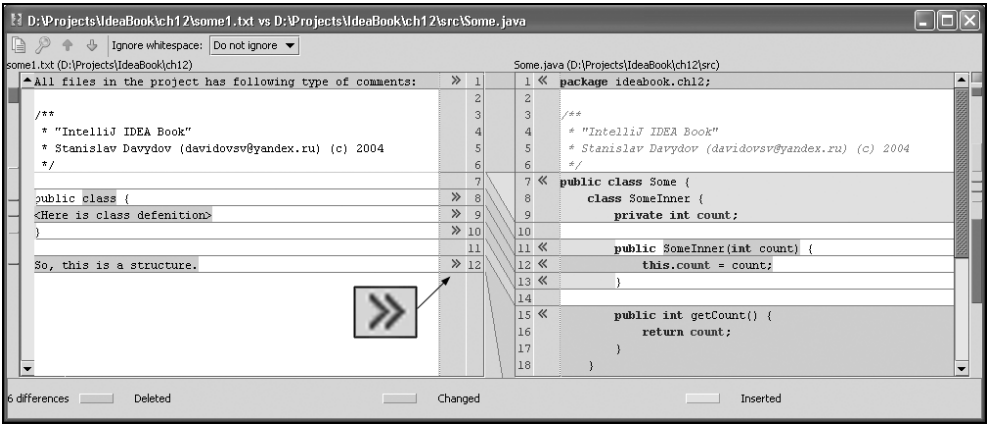


Рис. 12.1. Результаты сравнения двух файлов утилитой Diff

Окно сравнения содержит две панели, на которых представлено содержимое двух сравниваемых файлов (или частей файлов).

Обнаруженные изменения выделяются с помощью цветов:

- удаленные части (**Deleted**) — серый цвет;
- измененные части (**Changed**) — синий цвет;
- добавленные части (**Inserted**) — зеленый цвет;
- конфликтующие части (**Conflict**) — розовый цвет.

Данные цвета настраиваются с помощью диалога настроек цветов и шрифтов (*см. раздел 4.1.6.7*).

Цветовое выделение используется также и для обозначения соответствия строк в сравниваемых файлах. Между панелями расположено поле соответствия, в котором с помощью цветных полей представлено соответствие измененных строк.

12.1.4. Дополнительные команды

В правом поле файла, расположенного в левой панели сравнения, сразу перед номерами строк в местах измененных строк (**Changed**) находятся специальные командные символы '"', с помощью которых можно заменить изменившееся содержимое файла из правой панели (см. рис. 12.1).

Файлы, представленные на панелях, можно редактировать. При этом результаты сравнения будут автоматически изменяться в зависимости от редактируемого содержимого. Работают все команды редактирования (*см. раздел 3.6.4*).

В верхней части окна сравнения расположена панель инструментов со следующими командами:

- **Copy** (Копировать) (<Ctrl>+<C>) — копировать выделенное содержимое (*см. раздел 3.2.2.4*);
- **Find** (Поиск) (<Ctrl>+<F>) — поиск (*см. раздел 3.2.3.1*);
- **Move to Previous/Next Difference** (Переход к предыдущему/следующему изменению) (<Shift>+<F7>/<F7>);
- **Ignore whitespaces** (Игнорировать пробелы) — режим сравнения пробелов:
  - **Do not ignore** (Не игнорировать);
  - **Leading and trailing** (В начале и конце строк);
  - **All** (Все).

Утилита сравнения используется также при сравнении файлов, хранящихся в системах контроля версий. Там данная утилита имеет дополнительную функциональность, о которой мы расскажем в соответствующих разделах.

12.2. Локальная система контроля версий

Локальная система контроля версий хранит все изменения, произошедшие с файлами и директориями проекта в течение некоторого промежутка времени. По умолчанию изменения хранятся в течение трех дней, однако длительность хранения можно настроить с помощью диалога настройки локальной системы контроля версий (*см. раздел 4.1.12*).

Изменения в файлах и директориях могут фиксироваться в случае возникновения следующих событий:

- открытие проекта;
- компиляция:
  - проекта;
  - пакета;
  - файла.
- сборка проекта;
- запуск;
- отладка;
- успешное завершение тестов JUnit;
- завершение тестов JUnit с ошибками.

По умолчанию обрабатываются все эти события, хотя, при желании, некоторые из них можно выключить.

Локальная система контроля версий позволяет выполнять следующие действия:

- просматривать историю изменения файла с возможностью восстановления предыдущих версий;
- просматривать историю изменения директории с возможностью восстановления предыдущих версий;
- устанавливать метки на текущее состояние файлов и директорий с возможностью последующего отката к помеченной версии.

12.2.1. Версии файла

Для просмотра истории изменения файла необходимо выполнить команду **Local History | Show History** (Локальная история | Показать историю) из меню **Tools** (Инструменты) либо из контекстного меню файла, историю которого вы хотите посмотреть (рис. 12.2).

Для отображения истории изменений используется несколько модифицированная версия утилиты Diff (рис. 12.3).

Верхняя часть окна утилиты Diff осталась без значительных изменений (*см. раздел 12.1*), а в нижней части добавился список изменений файла.

Список изменений представлен в виде таблицы со следующими колонками:

- **Date** (Дата) — дата и время произошедшего изменения;

- **Version** (Версия) — название версии файла. В данной колонке могут быть значения трех типов:
  - пустые строки — зафиксированы обычные изменения файла;
  - именованные версии, например "Creating class" или "Running" — версии, полученные в результате каких-либо событий, произошедших с файлом или проектом;
  - именованные метки (*см. раздел 12.2.3*);
- **Action** (Действие) — действие, повлекшее изменения. В данной колонке также могут быть как пустые строки, обозначающие обычные изменения файла, а также именованные действия, например "Moving inner class", повлекшие изменения в результате рефакторинга, запуска внешних утилит и т. п.

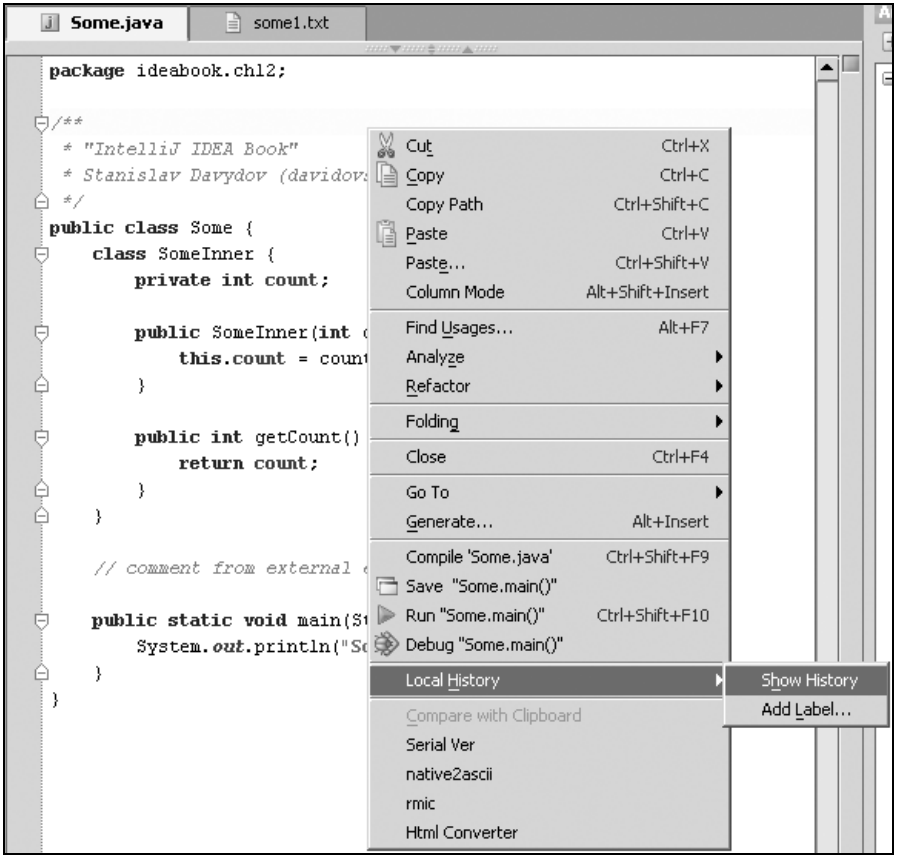


Рис. 12.2. Вызов команды просмотра истории изменений файла

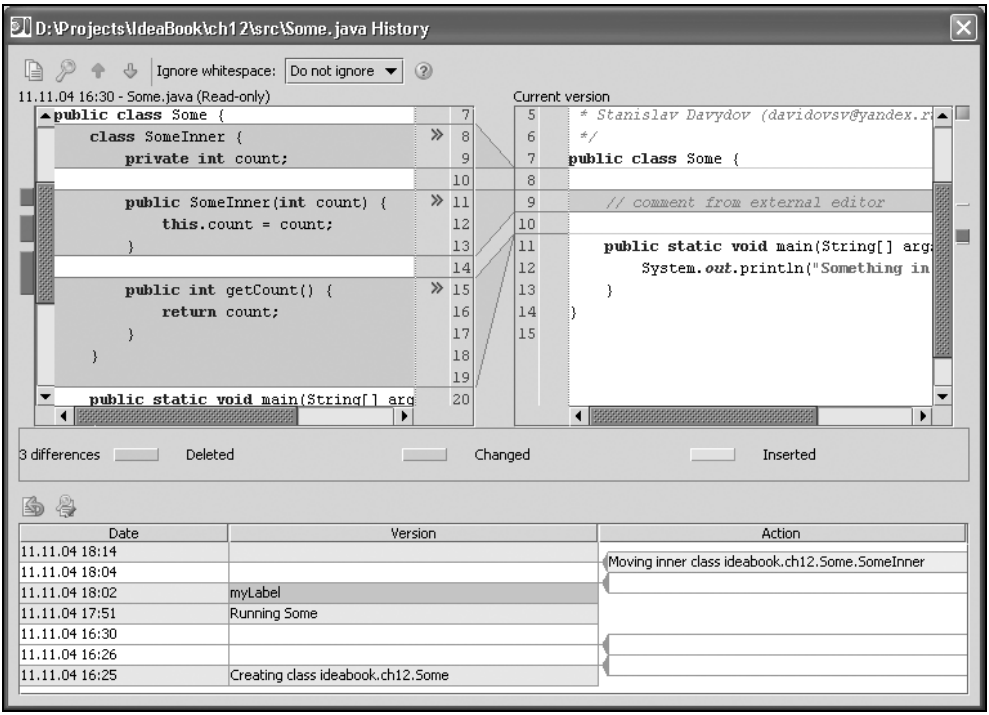


Рис. 12.3. Просмотр истории изменений файла

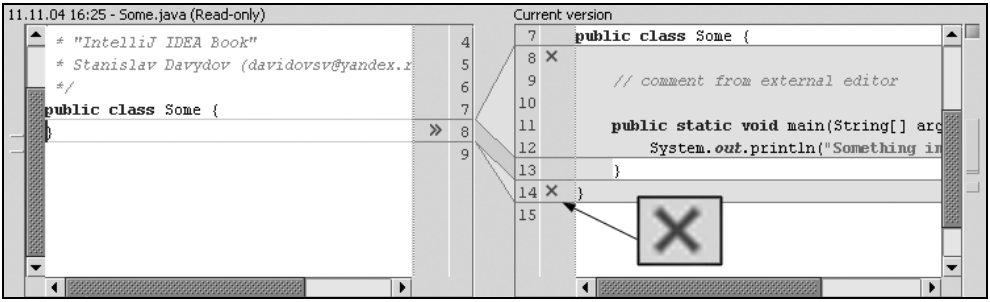


Рис. 12.4. Сравнений текущей версии файла с одной из предыдущих

Выделяя ячейку в колонке **Version** (Версия) или **Action** (Действие), можно просматривать изменения между текущей версией файла и той, которую вы выделили, либо той, которая была до выполнения действия (рис. 12.4).

При сравнении версий с текущей версией файла имеется возможность переносить и отменять изменения с помощью кнопок '**»**' (вставить изменение) и '**x**' (удалить изменение), размещенных рядом с помеченными изменениями.

Также имеется возможность сравнить любые две версии файла. Для этого нужно выделить в таблице версий две ячейки, соответствующие либо двум разным версиям, либо двум разным действиям.

В режиме просмотра истории изменений файла нельзя редактировать содержимое панелей сравнения, как это было возможно в обычном режиме сравнения файлов (*см. раздел 12.1*).

Над таблицей истории изменений файла расположена панель инструментов, в которой имеются две дополнительные команды:

- **Rollback to Revision/Action** (Откатить версию/действие) — команда отката изменений в файле к указанной версии файла либо к указанному действию, в зависимости от того, выделена ли в таблице ячейка из колонки **Version** (Версия) или **Action** (Действие);
- **Show Changes Only** (Показывать только изменения) — данная команда исключает из таблицы версии файла, которым не соответствуют изменения (рис. 12.5).

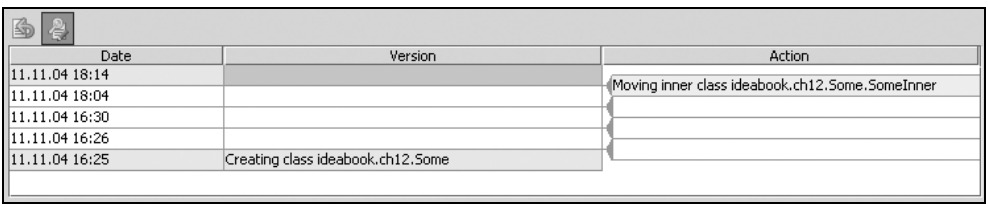


Рис. 12.5. Режим **Show Changes Only**

12.2.2. Версии директорий

История изменения директории хранит как изменения в конкретных файлах (*см. раздел 12.2.1*), так и изменения расположения файлов (рис. 12.6).

Таблица истории изменений аналогична той, что присутствует при просмотре истории изменений файла (*см. раздел 12.2.1*).

Верхняя часть окна занята двумя панелями, в которых сравниваются изменения, произошедшие в директории. Если выделена только одна ячейка в таблице истории изменений, сравнение производится с текущим состоянием директории. Если выделено две ячейки — сравнение производится между ними.

Изменения помечаются с помощью цветовых указателей, размещенных между панелями сравнения:

- синий указатель (**Modified**) — в файле произошли изменения;
- зеленый указатель (**Added**) — файл был добавлен;
- серый указатель (**Deleted**) — файл был удален.

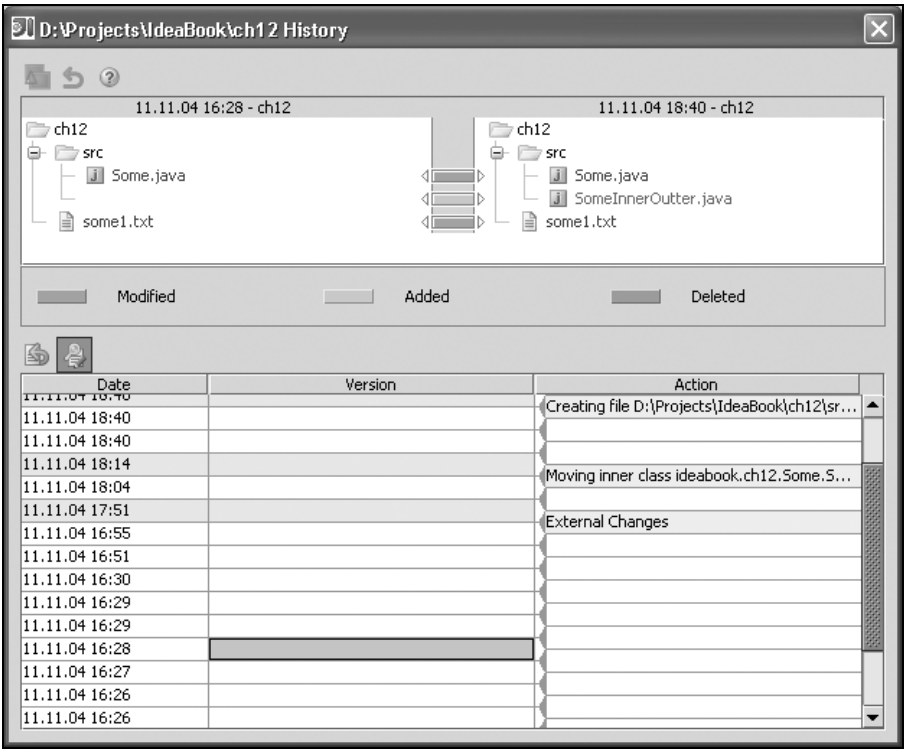


Рис. 12.6. История изменений директории

Цвета данных указателей настраиваются аналогично цветам изменений в файлах (*см. раздел 4.1.6*).

В верхней части окна просмотра изменений имеется панель инструментов, на которой расположены три команды:

- **Compare Versions** (Сравнить версии) — сравнить версии изменившегося файла;
- **Rollback** (Откатить) — откатить сделанные изменения;
- **Help** (Помощь) — открыть раздел встроенной системы помощи, посвященный просмотру изменений в локальной системе контроля версий.

12.2.3. Метки

В локальной системе контроля версий имеется два типа меток, которыми помечаются версии файлов:

- автоматические метки — метки, которые создаются при возникновении различных событий (например, компиляция, запуск и т. п.);
- метки, устанавливаемые самим пользователем.

Повлиять на создание автоматических меток можно только с помощью диалога настройки локальной системы контроля версий (*см. раздел 4.1.12*).

Второй тип меток можно установить с помощью команды **Local History | Add Label** (Локальная история | Установить метку). Пользовательские метки можно устанавливать как на отдельные файлы и директории, так и на весь проект целиком (рис. 12.7).

Рис. 12.7. Добавление метки

12.3. CVS

CVS — Concurrent Version System — одна из самых распространенных систем контроля версий. Свое распространение CVS получила из-за своих богатых возможностей, а также благодаря тому, что она является полностью бесплатной.

CVS используется в таких известных проектах с открытым кодом (open source), как Linux, Apache, проекты Sourceforge.org и многих других.

В IntelliJ IDEA имеется поддержка CVS, осуществляемая с помощью специального плагина CVS Integration. Подробнее о плагинах можно прочесть в *главе 15*, а сейчас мы расскажем об интеграции CVS в IntelliJ IDEA.

12.3.1. Настройки

Начнем наш рассказ об интеграции CVS в IntelliJ IDEA с настроек. В IntelliJ IDEA есть два уровня настроек CVS:

- глобальный уровень — настройки, актуальные для всех проектов в IntelliJ IDEA;
- уровень проекта — настройки CVS для конкретного проекта.

CVS является необязательной частью проекта. Проекты могут либо совсем не иметь связи с системой контроля версий, либо быть связанными с другими системами, например, SourceSafe или StarTeam. Поэтому команды

настройки CVS для проекта появляются в интерфейсе IntelliJ IDEA только тогда, когда для проекта активирована связь с CVS (*см. раздел 4.2.3*).

Рассмотрим настройки CVS для проекта.

12.3.1.1. Настройки для проекта

Диалог настроек CVS для проекта показан на рис. 12.8.

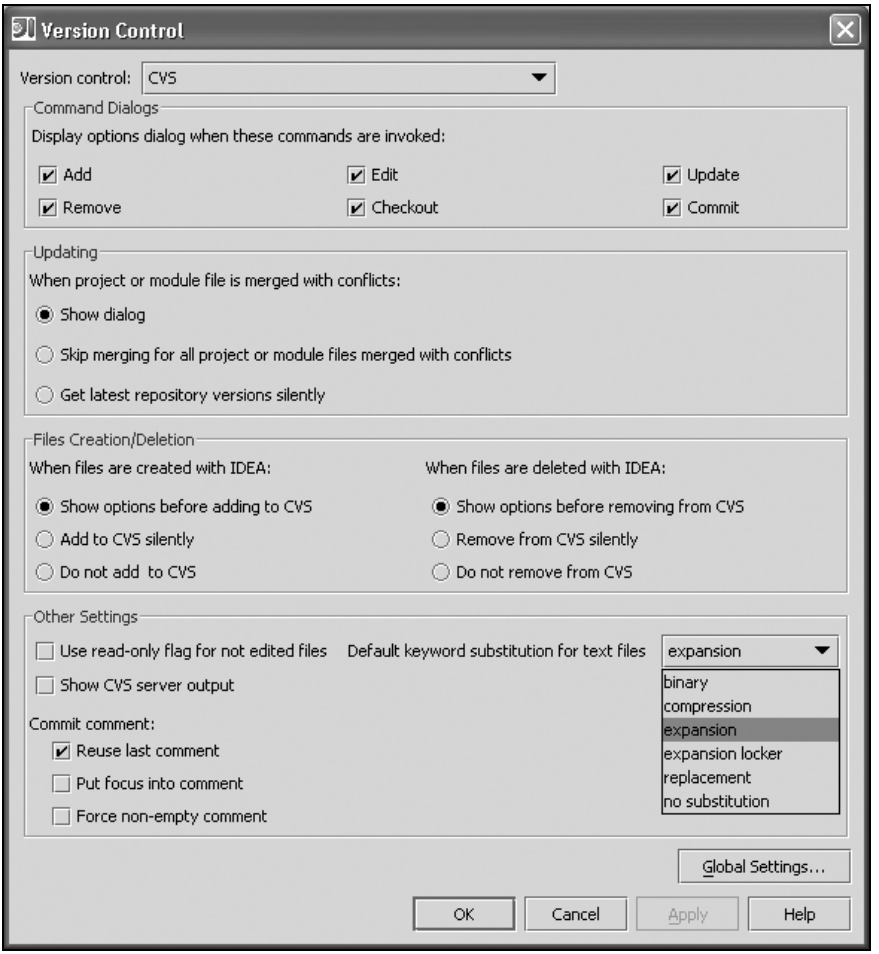


Рис. 12.8. Диалог настройки CVS для проекта

Опции в данном диалоге сгруппированы следующим образом:

- **Command Dialogs** (Диалоги команд) — опции отображения диалогов для следующих команд CVS:
  - **Add** (Добавить) — добавление файлов (*см. раздел 12.3.4.12*);

- **Remove** (Удалить) — удаление файлов;
- **Edit** (Редактировать) — пометить файл как редактируемый (*см. раздел 12.3.4.25*);
- **Checkout** (Получение изменений) — получение последней версии файла (*см. раздел 12.3.4.10*);
- **Update** (Обновление) — обновление файлов (*см. раздел 12.3.4.9*);
- **Commit** (Подтверждение) — подтверждение изменений в файлах (*см. раздел 12.3.4.7*);

❐ **Updating** (Обновление) — данная группа содержит значения опции **When project or module file is merged with conflicts** (Когда файл проекта или модуля был объединен с конфликтами):

- **Show dialog** (Показать диалог) — отображать диалог с возможными вариантами действий;
- **Skip merging for all project or module files merged with conflicts** (Не производить объединений для всего файла проекта или модуля);
- **Get latest repository versions silently** (Взять последнюю версию файла из репозитория);

❐ **Files Creation/Deletion** (Создание и удаление файлов) — в данной группе содержатся настройки двух опций создания и удаления файлов. IntelliJ IDEA имеет возможность автоматически добавлять в CVS файлы, вновь созданные в проекте, а также удалять из CVS файлы, удаленные из проекта:

- **When files are created with IDEA** (Когда файлы создаются с помощью IntelliJ IDEA):
  - ◊ **Show options before adding to CVS** (Отображать диалог с возможными вариантами действий перед добавлением в CVS);
  - ◊ **Add to CVS silently** (Добавлять файлы в CVS без дополнительных вопросов);
  - ◊ **Do no add to CVS** (Не добавлять файлы в CVS);
  - ◊ **When files are deleted with IDEA** (Когда файлы удаляются с помощью IntelliJ IDEA):
  - ◊ **Show options before removing from CVS** (Отображать диалог с возможными вариантами действий перед удалением из CVS);
  - ◊ **Remove from CVS silently** (Удалять из CVS автоматически);
  - ◊ **Do not remove from CVS** (Не удалять из CVS);

❐ **Other Settings** (Прочие настройки):

- **Use read-only flag for not edited files** (Помечать флажком "только для чтения" файлы, не открытые для редактирования) (*см. раздел 12.3.4.25*);

- **Show CVS server output** (Показывать вывод результата команд с CVS-сервера);
  - ◊ **Commit comment** (Комментарий при подтверждении изменений):
  - ◊ **Reuse last comment** (Использовать последний комментарий);
  - ◊ **Put focus into comment** (Помещать курсор ввода на поле добавления комментария);
  - ◊ **Force non-empty comment** (Не открывать дополнительный диалог, если имеется непустой комментарий). Если комментария нет, IntelliJ IDEA будет выводить отдельный запрос на добавления изменений с пустым комментарием;
- **Default keyword substitution for text files** (Подстановка ключевых слов подстановки для текстовых файлов по умолчанию). При работе с CVS с помощью утилит командной строки данные ключевые слова задаются в виде дополнительной опции при получении и обновлении файлов из CVS (подробнее см. **http://www.cvshome.org**):
  - ◊ binary (опция CVS -kb) — аналогично "no substitution", но также дополнительно поддерживается сохранение переводов строк, что важно для бинарных файлов;
  - ◊ compression (опция CVS -kk) — создавать только имена ключевых слов без указания конкретных значений;
  - ◊ expansion (опция CVS -kkv) — создавать строки по ключевым словам с использованием стандартной формы;
  - ◊ expansion locker (опция CVS -kkvl) — аналогично "expansion", но также добавляет имя пользователя, заблокировавшего файл;
  - ◊ replacement (опция CVS -kv) — создавтаь только значения без ключевых слов;
  - ◊ no substitution (опция CVS -ko) — создавать старую строку с ключевым словом перед загрузкой файла в репозиторий (не подставлять новое значение).

12.3.1.2. Глобальные настройки

В правом нижнем углу диалога настройки (см. рис. 12.8) находится кнопка вызова диалога настройки глобальных опций CVS (рис. 12.9).

В данном диалоге содержатся следующие опции и параметры:

- ❐ **Use UTF-8 encoding** (Использовать кодировку UTF-8);
- ❐ **Use gzip compression** (Использовать сжатие gzip);
- ❐ **Password file** (Расположение файла с паролем доступа к CVS);
- ❐ **Connection timeout** (Время простоя соединения с сервером) (в секундах).

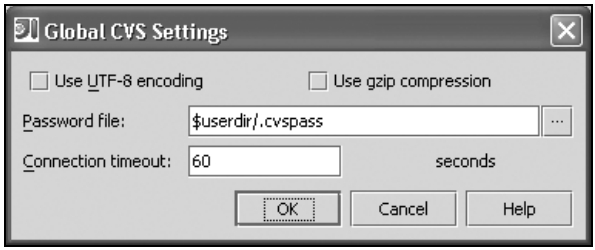


Рис. 12.9. Глобальные настройки CVS

12.3.1.3. Репозиторий CVS (CVS Roots)

Репозиторий CVS (CVS Roots) — это путь к репозиторию на CVS-сервере, где хранятся версии файлов.

Для работы с CVS необходимо, чтобы для файлов проекта был задан такой репозиторий (он может быть и один на всех).

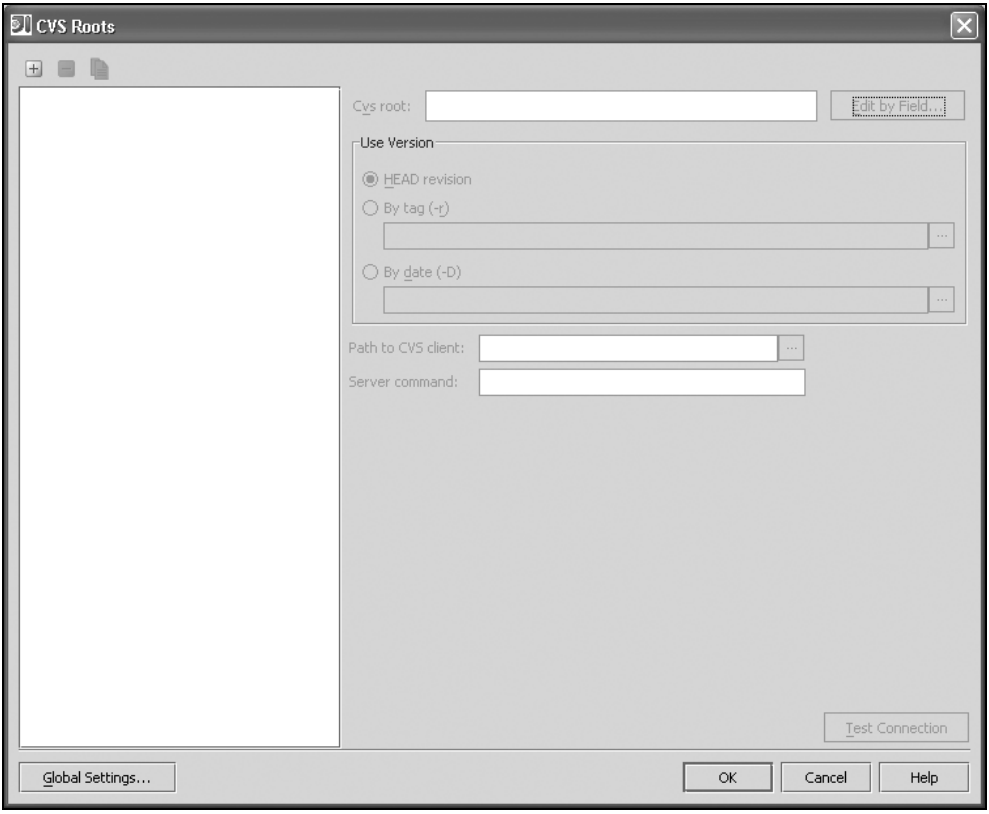


Рис. 12.10. Конфигурация репозиториев CVS

Настройка репозиториев CVS производится с помощью команды **CVS | Configure CVS Roots** (CVS | Конфигурировать репозитории CVS) главного меню раздела CVS (данный раздел появляется в меню, если для проекта включена поддержка CVS (*см. раздел 12.3.1.1*)).

Команда настройки репозиториев CVS открывает диалог, представленный на рис. 12.10.

С помощью команд, расположенных на панели инструментов в верхней части данного диалога, можно добавлять, удалять и копировать репозитории CVS.

Рассмотрим добавление репозитория.

Нажмем кнопку + на панели инструментов, при этом поля в правой части диалога станут активными, и мы сможем задать настройки данного репозитория (рис. 12.11).

Для репозитория необходимо настроить его путь (**Cvs root**) — поле в правой верхней части диалога. Путь можно записать в виде строки, либо настроить с помощью диалога, открывающегося при нажатии на кнопку **Edit by Field** (Редактировать по полям) (рис. 12.12).

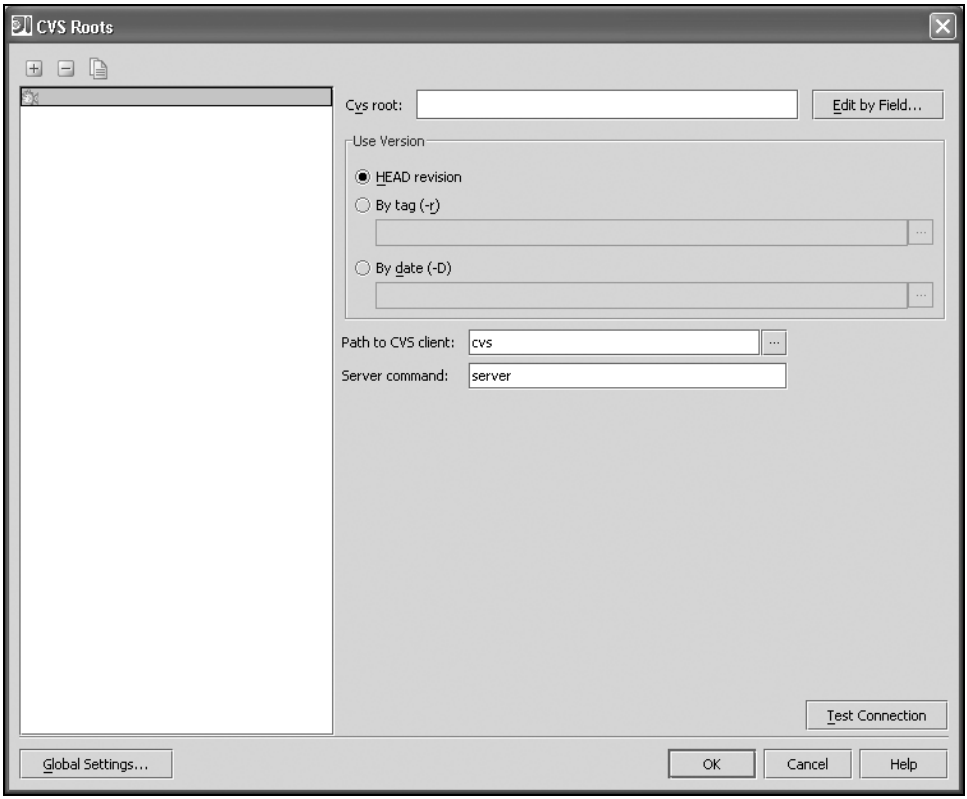


Рис. 12.11. Добавление репозитория

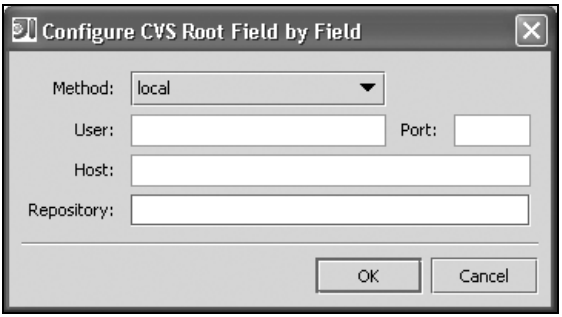


Рис. 12.12. Настройка пути репозитория

Путь репозитория состоит из следующих полей:

- **Method** (Метод) — метод работы с сервером. IntelliJ IDEA поддерживает следующие методы работы с CVS:
  - `pserver`;
  - `ext`;
  - `ssh` (internal implementation) (внутренняя имплементация протокола ssh);
  - `local`;
- **User** (Пользователь) — имя пользователя;
- **Port** (Порт) — номер порта;
- **Host** (Хост) — имя хоста сервера;
- **Repository** (Репозиторий) — путь репозитория на сервере.

В зависимости от метода работы с CVS, в диалоге настройки репозиториев CVS могут добавляться некоторые дополнительные параметры и свойства. Мы рассмотрим эти свойства для каждого метода в отдельности.

Общие параметры

К общим параметрам репозиториев относится параметр **Use Version** (Использовать версию) — использовать следующую версию файлов в репозитории:

- **HEAD revision** (Основную версию);
- **By tag** (Версия по тегу из CVS);
- **By date** (Версия по дате).

Дополнительно в нижней части диалога имеется кнопка **Test Connection** (Тестировать соединение).

Метод *pserver*

Репозиторий, доступный с помощью метода `pserver`, имеет следующие настройки (рис. 12.13).

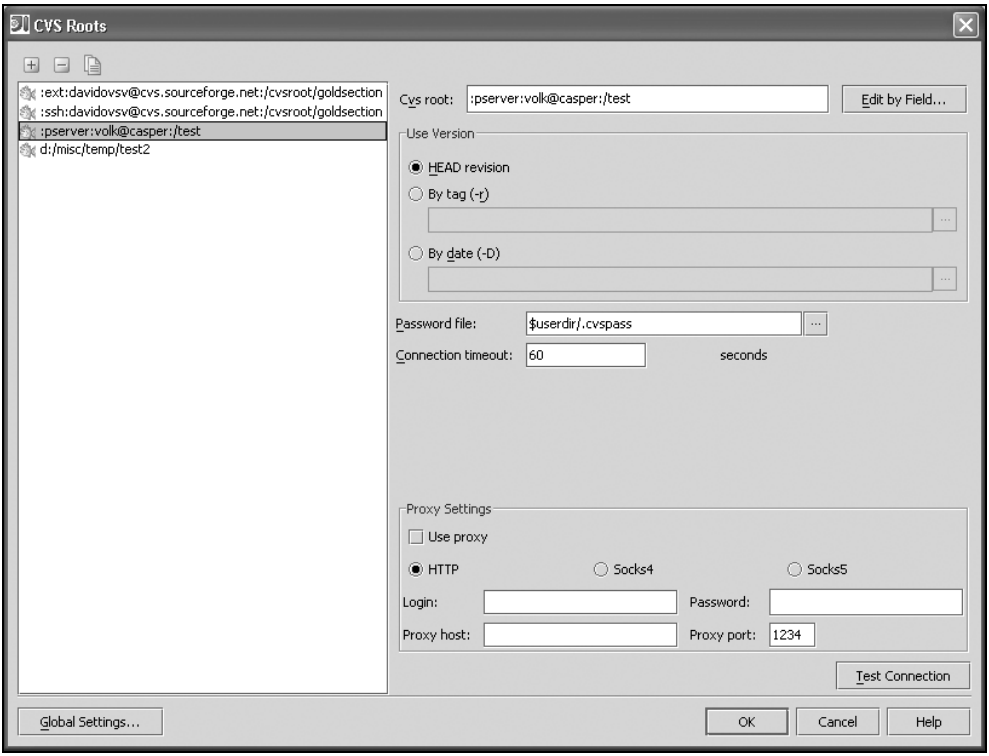


Рис. 12.13. Настройки репозитория для метода `pserver`

Дополнительно к общим настройкам для репозитория с методом `pserver` добавились следующие:

- **Password file** (Расположение файла с паролем доступа к CVS);
- **Connection timeout** (Время простоя соединения с сервером) (в секундах);
- **Proxy Settings** (Настройки прокси):
  - **Use proxy** (Использовать прокси) — тип прокси:
    - HTTP;
    - Socks4;
    - Socks5;
  - **Login** (Логин);
  - **Password** (Пароль);
  - **Proxy host** (Имя хоста прокси);
  - **Proxy port** (Номер порта прокси).

Метод *ext*

Данный метод использует внешнюю команду для осуществления связи (например, rsh или ssh) с сервером. Настройки репозитория с данным методом доступа представлены на рис. 12.14.

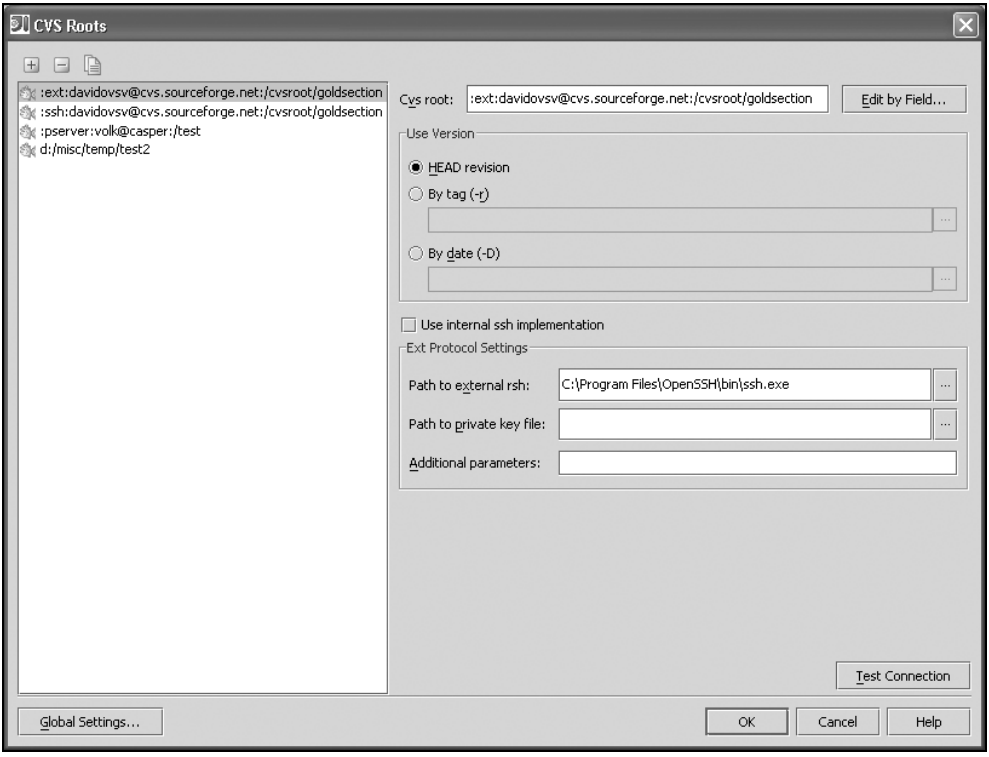


Рис. 12.14. Настройки репозитория для метода `ext`

Для данного метода имеется возможность использовать внутреннюю имплементацию протокола SSH (**Use internal ssh implementation**). Если данная опция включена, дополнительные настройки репозитория становятся аналогичными настройкам метода `ssh` (*см. следующий раздел "Метод ssh"*).

Для данного метода доступа имеются следующие настройки (**Ext Protocol Settings**):

- **Path to external rsh** (Путь к внешней программе rsh);
- **Path to private key file** (Путь к файлу с закрытым ключом);
- **Additional parameters** (Дополнительные параметры) — дополнительные параметры для передачи внешней программе.

Метод *ssh*

Данный метод использует внутреннюю имплементацию протокола SSH и не требует наличия внешних утилит для работы по этому протоколу.

Настройки для данного метода представлены на рис. 12.15.

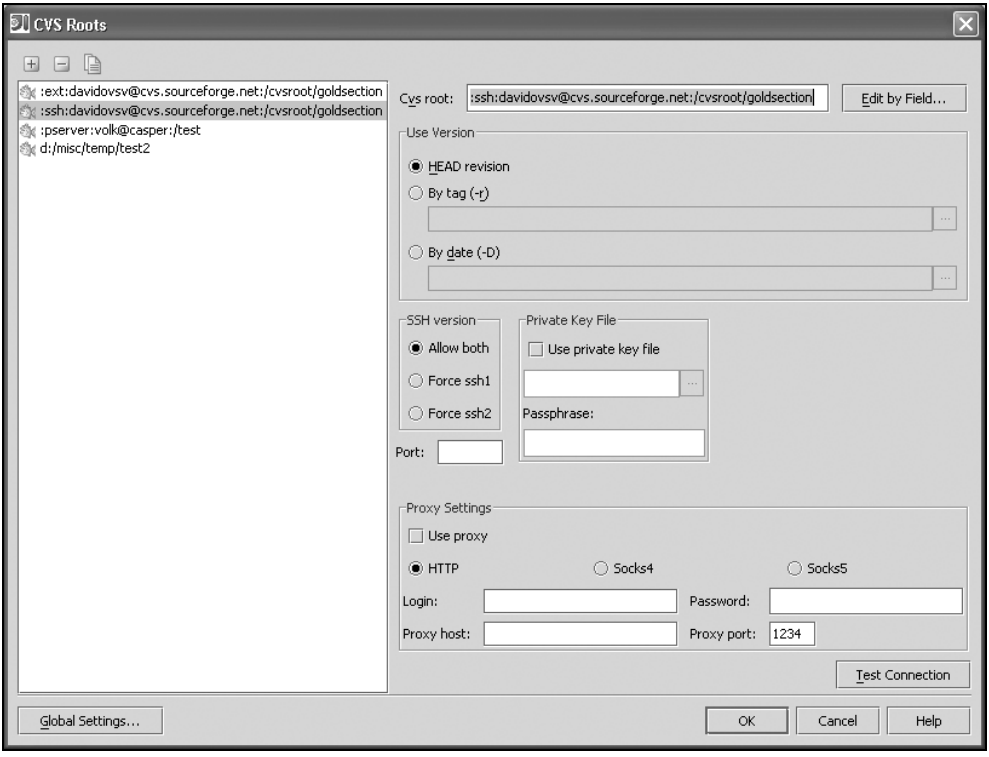


Рис. 12.15. Настройки репозитория для метода `ssh`

Дополнительные настройки для данного метода:

- **SSH version** (Версия протокола SSH):
  - **Allow both** (Разрешить обе) — разрешить работать с обеими версиями (ssh1 и ssh2);
  - **Force ssh1** (Только ssh1);
  - **Force ssh2** (Только ssh2);
- **Private Key File** (Файл с закрытым ключом) — данная группа настроек задает путь к файлу с закрытым ключом, его пароль (**Passphrase**), а также опцию, которая разрешает использовать закрытый ключ (**Use private key file**);
- **Port** (Порт) — номер порта для подсоединения.

Для данного метода также доступны настройки прокси, аналогичные настройкам для метода `pserver`.

Метод *local*

Данный метод имеет две дополнительные настройки (рис. 12.16).

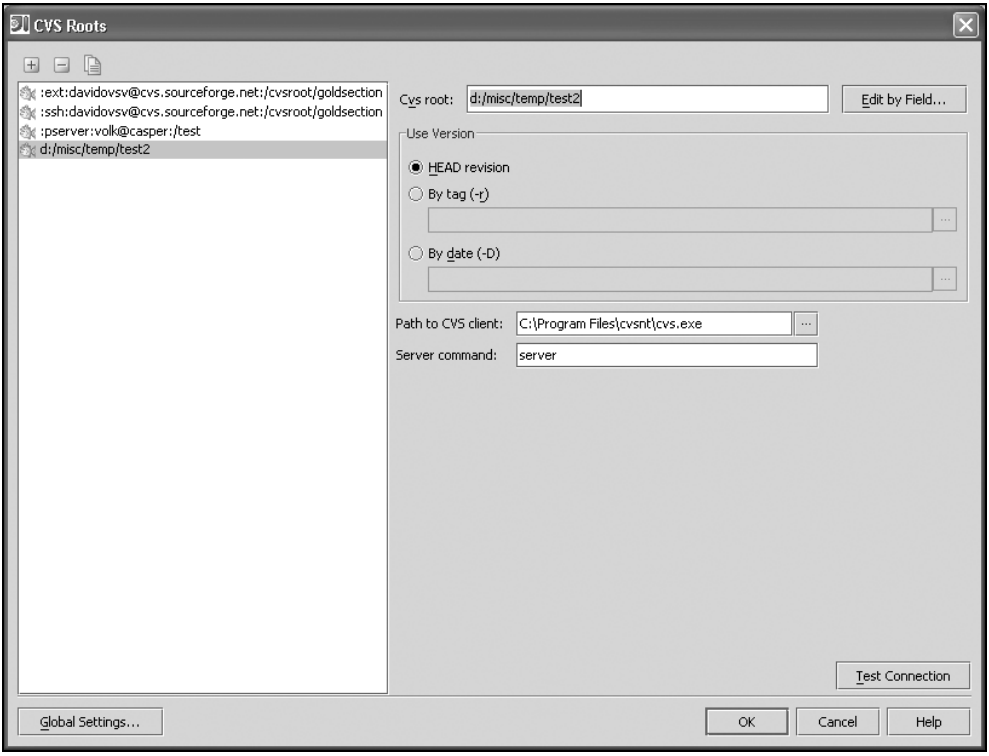


Рис. 12.16. Настройки репозитория для метода `local`

Настройки для метода `local`:

- **Path to CVS client** (Путь к программе-клиенту CVS);
- **Server command** (Команда сервера).

12.3.2. Работа с проектом

Чтобы файлы проекта оказались связанными с репозиторием системы контроля версий, необходимо выполнить одно из следующих действий:

- если в CVS уже хранится проект и все его файлы, проект нужно оттуда скачать;
- если проект новый и в CVS еще не хранится, его нужно туда сохранить или импортировать.

Эти действия необходимы, поскольку CVS содержит информацию о файлах в специальных директориях (CVS/), расположенных в каждой директории с локальными файлами.

Если вы уже имеете локальную структуру файлов CVS (это возможно, если вы используете утилиты командной строки для работы с CVS-репозиторием), никаких дополнительных действий для начала работы с CVS в IntelliJ IDEA не требуется.

Мы рассмотрим импорт проекта в CVS, а затем скачивание проекта из CVS.

12.3.2.1. Импорт проекта в CVS

Импорт проекта в CVS осуществляется с помощью команды **Import into CVS** (Импорт в CVS), расположенной в меню **File** (Файл) (*см. раздел 3.2.1.18*).

Данная команда запускает мастер импорта в CVS (рис. 12.17).

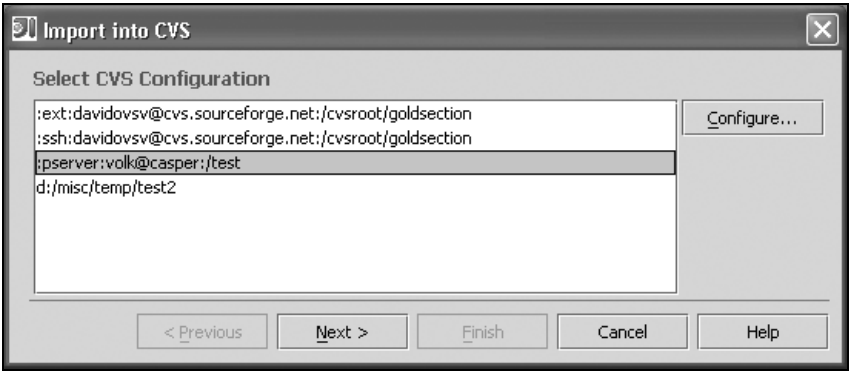


Рис. 12.17. Мастер импорта в CVS

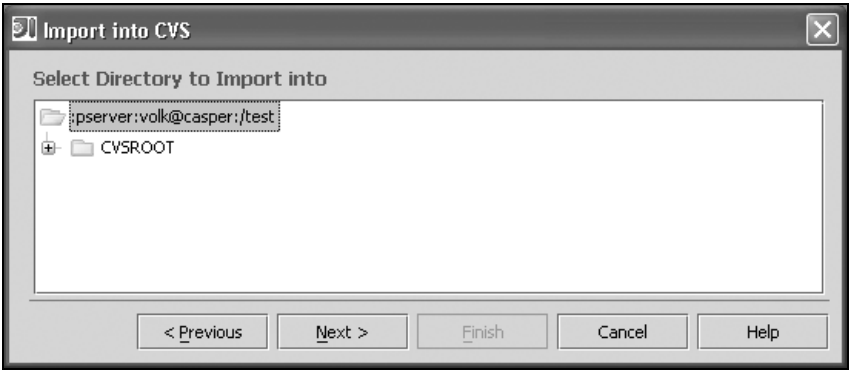


Рис. 12.18. Выбор директории, в которую будет произведен импорт, из CVS-репозитория

Чтобы импортировать проект в CVS, необходимо выбрать конфигурацию репозитория CVS, в который будет производиться импорт. Подробнее про конфигурацию репозитория написано в *разделе 12.3.1.3*. Выберем требуемую конфигурацию и перейдем к следующему шагу (рис. 12.18).

На следующем шаге из CVS-репозитория необходимо выбрать директорию, в которую будет произведен импорт нашего проекта. Мы выбираем корневую директорию, поскольку другой директории в нашем репозитории нет (CVSROOT — служебная директория CVS).

Далее следует выбрать директорию проекта, который мы хотим импортировать в CVS (рис. 12.19).

Затем у нас есть возможность выбрать соответствие расширений файлов нашего проекта способу хранения этих файлов в репозитории (рис. 12.20).

В данном списке присутствуют все файлы, расширения которых известны IntelliJ IDEA (*см. раздел 4.1.11*).

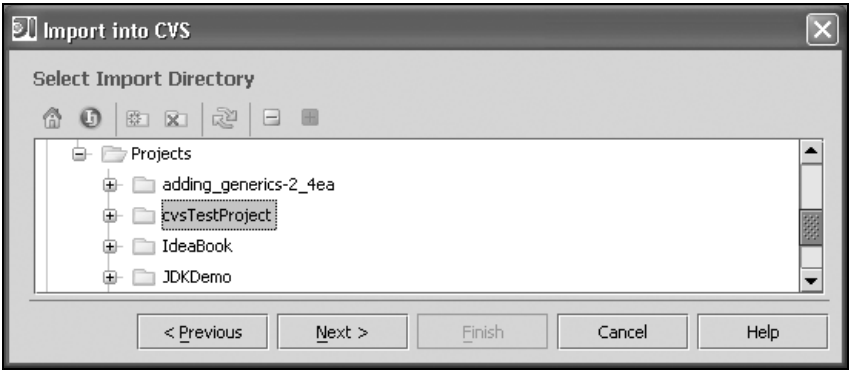


Рис. 12.19. Выбор директории проекта для импорта

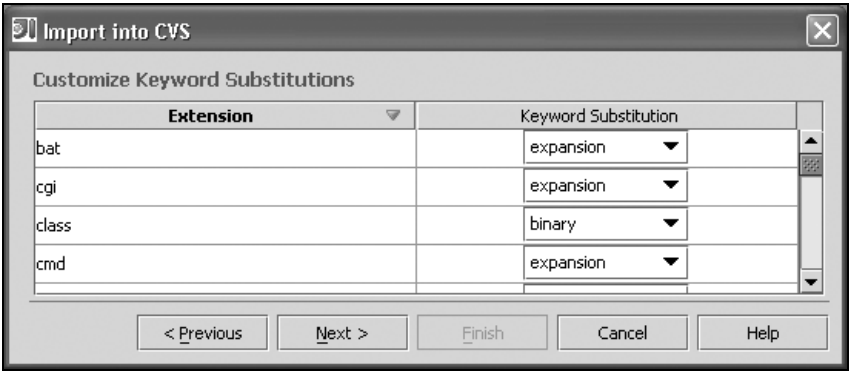


Рис. 12.20. Настройка соответствия расширений файлов проекта способу хранения файлов в репозитории

По умолчанию все текстовые файлы имеют тип `expansion`, а бинарные — `binary`.

На заключительном шаге можно задать следующие опции (рис. 12.21):

- **Name in repository** (Имя в репозитории) — имя папки в репозитории;
- **Vendor** (Производитель) — имя поставщика проекта. В данном имени не должно быть пробелов и знаков пунктуации;
- **Release tag** (Тег выпуска) — некоторое имя, которое обозначает название версии, импортируемой в CVS;
- **Log message** (Сообщение в лог);
- **Checkout after import** (Загрузить из репозитория после импорта) — загрузить файлы проекта из репозитория после импорта;
- **Make checked out files read-only** (Пометить загруженные файлы как доступные только для чтения).

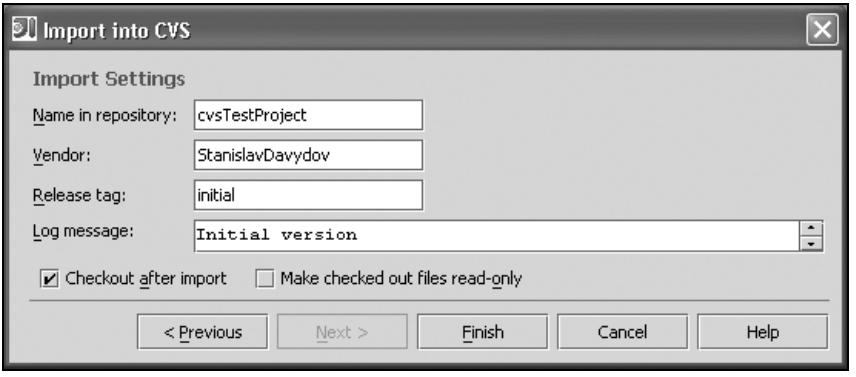


Рис. 12.21. Заключительный шаг импорта проекта

Если вы включили опцию **Checkout after import** (Загрузить из репозитория после импорта) — проект после импорта будет автоматически скачен из CVS. Если данная опция была выключена, для дальнейшей работы CVS с файлами проекта необходимо загрузить проект из CVS.

12.3.2.2. Загрузка проектов из CVS

Загрузка проектов из CVS осуществляется с помощью команды **Check out from CVS** (Закачать из CVS) из меню **File** (Файл), которая запускает мастер загрузки проекта из CVS (рис. 12.22).

На первом шаге необходимо выбрать репозиторий для загрузки. Затем нужно из репозитория выбрать директорию, в которой находятся файлы проекта (рис. 12.23).

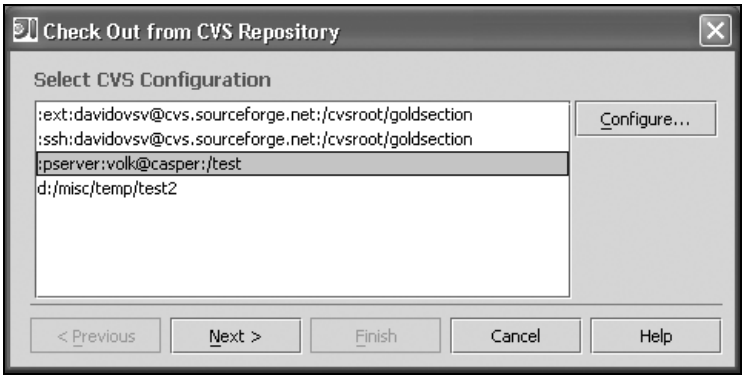


Рис. 12.22. Мастер загрузки из CVS: выбор репозитория для загрузки

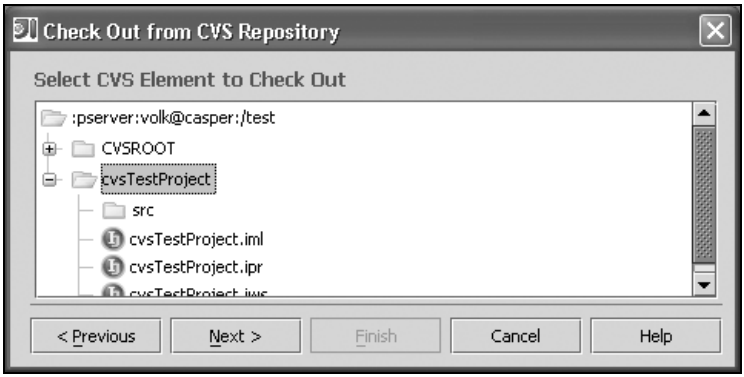


Рис. 12.23. Выбор из репозитория директории с файлами проекта

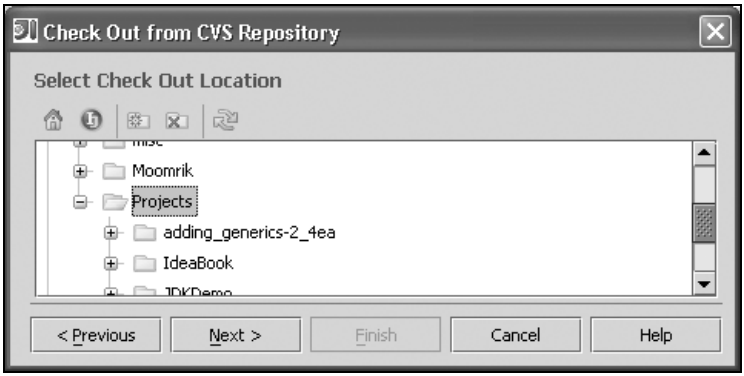


Рис. 12.24. Выбор директории на диске, куда будут загружены файлы из CVS

Далее нужно выбрать директорию на диске, куда будут загружены файлы из CVS (рис. 12.24).

На заключительном шаге нужно выбрать один из вариантов размещения файлов. Кроме того, возможно задать для них несколько дополнительных опций (рис. 12.25).

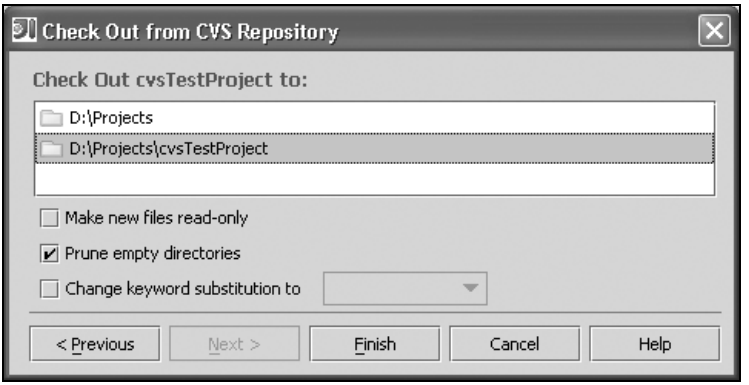


Рис. 12.25. Заключительный этап загрузки проекта из репозитория

Имеется возможность задать следующие опции:

- **Make new files read-only** (Пометить новые файлы как доступные только для чтения);
- **Prune empty directories** (Сокращать пустые директории);
- **Change keyword substitution to** (Изменить подстановку ключевого слово на):
  - binary (-kb);
  - compression (-kk);
  - expansion (-kkv);
  - expansion locker (-kkvl);
  - replacement (-kv);
  - no substitution (-ko).

После того как проект загружен из CVS, для файлов проекта доступны все команды работы с CVS (*см. раздел 12.3.4*).

12.3.2.3. Просмотр репозитория CVS

С помощью команды **Browse CVS Repository** (Просмотр репозитория CVS) можно просматривать содержимое выбранного репозитория CVS (рис. 12.26).

После подтверждения выбора будет открыто инструментальное окно CVS, в котором будет отображено содержимое выбранного репозитория (рис. 12.27).

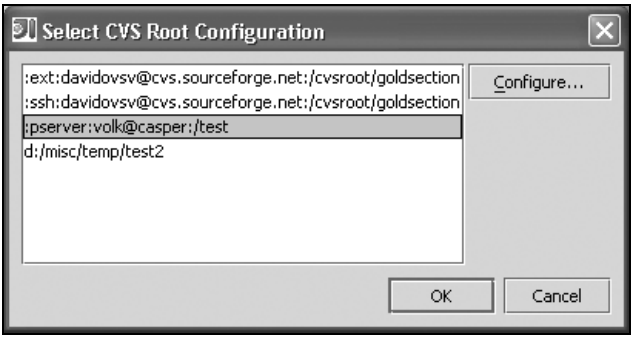


Рис. 12.26. Выбор репозитория для просмотра

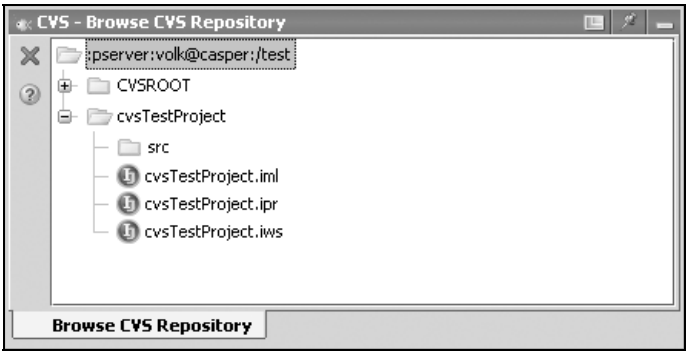


Рис. 12.27. Просмотр репозитория CVS

Для файлов и директорий в окне просмотра доступны следующие команды из контекстного меню:

- **Jump to Source** (Переход к исходному тексту) — данная команда открывает в редакторе IntelliJ IDEA файл из CVS. Данный файл не доступен для редактирования, даже если этот файл имеется в проекте. Дело в том, что в режиме просмотра репозитория IntelliJ IDEA никоим образом не связывает файлы из репозитория с файлами проекта, т. е. просматривать можно абсолютно произвольный репозиторий;
- **Checkout** (Загрузить) — загрузить из репозитория (*см. раздел 12.3.4.10*);
- **Show History** (Показать историю изменений) — команда открывает таблицу с историей изменений (*см. раздел 12.3.4.19*);
- **Annotate** (Аннотация) — команда открывает панель с аннотациями к данному файлу (*см. раздел 12.3.4.21*).

12.3.3. Визуальное выделение файлов

Файлы проекта, связанного с системой контроля версий, помечаются разными цветами в зависимости от того, какой статус они имеют в репозитории. Всего есть девять таких цветовых статусов:

- коричневый (**Unknown**) — статус файла неизвестен (такой статус имеют файлы проекта, которые не добавлены в CVS);
- сиреневый (**Merged**) — файл был объединен с версией из репозитория;
- фиолетовый (**Deleted from file system**) — файл был удален из файловой системы, но все еще есть в CVS;
- серый (**Deleted**) — файл удален из CVS;
- бурый (**Ignored**) — файл не участвует в операциях CVS;
- синий (**Modified**) — файл был изменен локально;
- зеленый (**Added**) — файл был добавлен в CVS, но добавление еще не было подтверждено;
- красный (**Merged with conflicts**) — файл был объединен с версий из CVS, и при этом возникли конфликты;
- черный (**Up to date**) — локальный файл имеет последнюю версию.

Настройка цвета файлов с разными статусами производится с помощью диалога настройки шрифтов и цветов (*см. раздел 4.1.6*).

12.3.4. Команды

Команды для работы с CVS собраны в разделе **CVS** главного меню. Данный раздел присутствует в меню, только если для текущего проекта настроено использование CVS в качестве системы контроля версий.

12.3.4.1. *Show File View (Показать окно просмотра файлов)*

Данная команда открывает инструментальное окно **CVS File View** (Просмотр файлов CVS) (рис. 12.28), в котором отображается файловая структура проекта и дополнительная информация о файлах из CVS.

В данном окне имеется инструментальная панель, две панели с настройками отображения информации, а также панель со структурой проекта, которая может отображаться в двух режимах: в виде иерархической структуры (см. рис. 12.28) и в виде плоской табличной структуры (рис. 12.29).

На панели инструментов представлены следующие команды:

- **Show as Tree** (Показывать как дерево) — команда-переключатель режима просмотра между иерархическим и плоским табличным;
- **Refresh View** (Обновить вид);

- **Expand/Collapse All** (Развернуть/Свернуть все) — данные команды работают только в режиме иерархического просмотра. Они позволяют развернуть всю иерархию файлов и директорий, а также свернуть ее;
- **Close** (Закрыть) — команда закрывает инструментальное окно **CVS File View** (Просмотр файлов CVS);
- **Help** (Помощь) — команда открывает окно встроенной справочной системы.

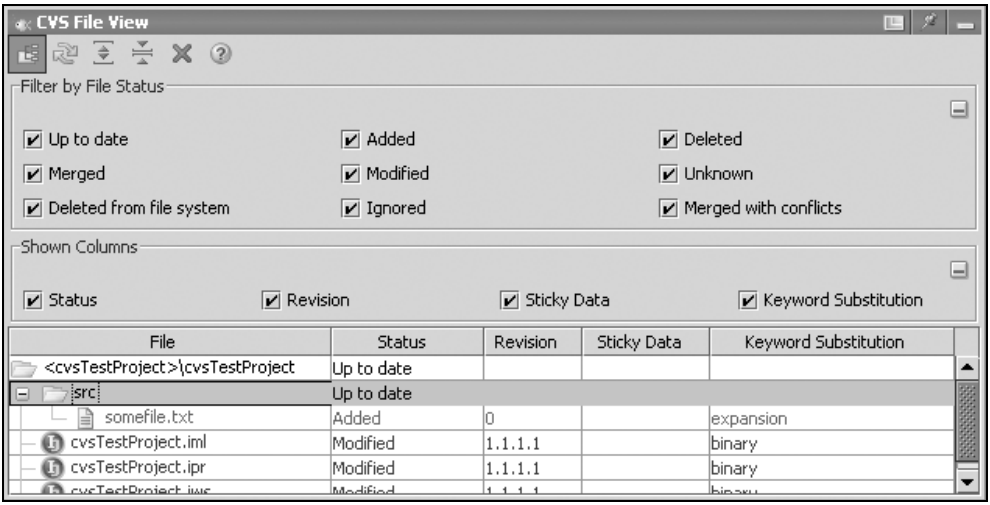


Рис. 12.28. Инструментальное окно **CVS File View**

File	Path	Status	Revision	Sticky Data	Keyword Substitution
cvsTestProject.iml	cvsTestProject.iml	Modified	1.1.1.1		binary
cvsTestProject.ipr	cvsTestProject.ipr	Modified	1.1.1.1		binary
cvsTestProject.iws	cvsTestProject.iws	Modified	1.1.1.1		binary

Рис. 12.29. Табличная форма представления информации о файлах

Существует ряд дополнительных опций, объединенных в группы:

- **Filter by File Status** (Показывать файлы со статусом):
  - **Up to date** (Имеющие текущую версию);
  - **Merged** (Объединенные);
  - **Deleted from file system** (Удаленные локально);
  - **Added** (Добавленные);
  - **Modified** (Измененные);

- **Ignored** (Игнорируемые);
- **Deleted** (Удаленные из CVS);
- **Unknown** (Неизвестные) — файлы, не добавленные в CVS;
- **Merged with conflicts** (Объединенные с конфликтами);

❒ **Shown Columns** (Показывать колонки в таблице):

- **Status** (Статус);
- **Revision** (Версию);
- **Sticky Data** (Ненужные ("липкие") данные) — названия тегов или веток, которые "приклеились" к файлу;
- **Keyword Substitution** (Подстановку ключевых слов).

12.3.4.2. *Commit Project (Подтвердить изменения в проекте)*

"Горячие" клавиши: <Ctrl>+<K>.

Команда открывает диалог подтверждения изменений для файлов всего проекта (рис. 12.30).

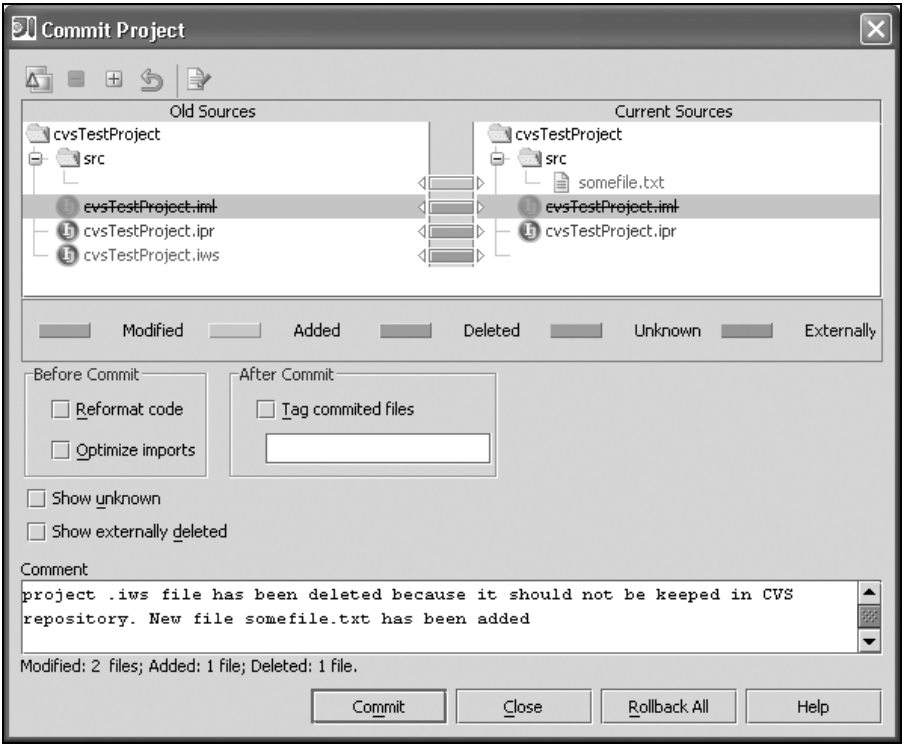


Рис. 12.30. Диалог **Commit Project**

В верхней части диалога расположена панель инструментов, под ней находятся две панели, аналогичные тем, что представлены в диалоге сравнения изменений, произошедших в директориях (*см. раздел 12.2.2*). Под этими панелями расположено несколько групп опций, а в самом низу диалога — управляющие кнопки.

На панели инструментов содержатся следующие команды:

- **Show changes** (Показать изменения) — данная команда открывает диалог сравнения версий файлов, полностью идентичный тому, о котором мы рассказывали в *разделе 12.2.1* с небольшим дополнением (рис. 12.31).

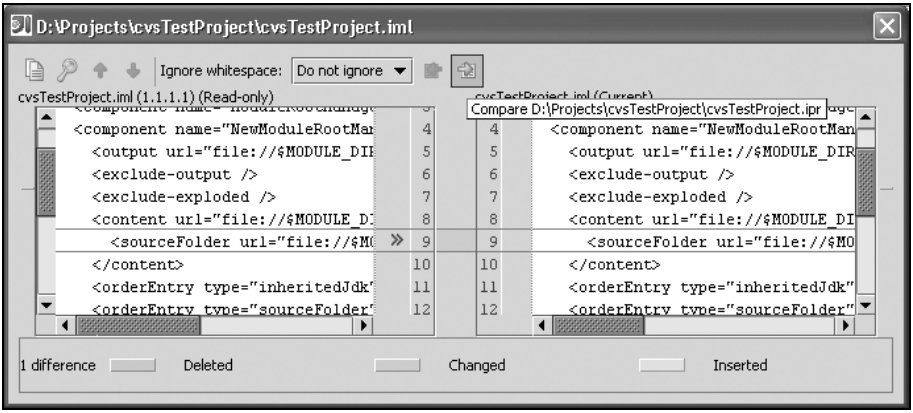


Рис. 12.31. Диалог сравнения двух версий

На панели инструментов диалога расположены две дополнительные кнопки, которые позволяют, не закрывая диалога, перемещаться по файлам из списка (см. рис. 12.30);

- **Exclude from commit** (Исключить из подтверждения) — исключить отдельный файл или все файлы в директории из подтверждения изменений. Имена таких файлов будут перечеркнуты (см. рис. 12.30);
- **Rollback Local Changes** (Откатить локальные изменения) — команда отменяет все изменения, выполненные в файле локально;
- **Edit Source** (Редактировать исходный текст) (<F4>) — команда открывает в редакторе указанный файл.

В диалоге подтверждения изменений для всего проекта содержатся следующие опции:

- **Before Commit** (Перед подтверждением) — группа опций, указывающих необходимые действия перед подтверждением изменений:
  - **Reformat code** (Переформатировать код) — переформатировать код подтверждаемых файлов (*см. раздел 3.2.11.1*);
  - **Optimize imports** (Оптимизировать импорты) (*см. раздел 3.2.11.2*);

- **After Commit** (После подтверждения) — команды, которые нужно выполнить после подтверждения изменений:
  - **Tag commited files** (Пометить подтвержденные файлы) — опция позволяет установить тег на файлы, изменения в которых были подтверждены;
- **Show unknown** (Показывать неизвестные) — опция включает режим отображения файлов, статус которых неизвестен (*см. раздел 12.3.3*);
- **Show externally deleted** (Показывать удаленные извне) — показывать файлы, которые были удалены извне;
- **Comment** (Комментарий) — комментарий к изменениям.

В данном диалоге имеются также четыре управляющие кнопки в нижней части окна диалога:

- **Commit** (Подтвердить) — подтвердить изменения;
- **Close** (Закрыть) — закрыть диалог без подтверждения;
- **Rollback All** (Откатить все) — откатить все изменения;
- **Help** (Помощь) — открыть справочную систему.

12.3.4.3. *Update Project (Обновить проект)*

Данная команда позволяет проверить обновления файлов во всем проекте. Команда удобна тем, что автоматически может проверять изменения для разных корневых директорий CVS.

Команда открывает диалог, представленный на рис. 12.32.

В данном диалоге представлены следующие опции:

- **Branch Merging** (Объединение веток):
  - **Don't merge** (Не объединять);
  - **Merge with branch (-j)** (Объединить с веткой);
  - **Merge two branches (-j -j)** (Объединить две ветки);
- **Use Version** (Использовать версию):
  - **Default** (По умолчанию);
  - **By tag (-r)** (По тегу);
  - **By date (-D)** (По дате);
- **Reset sticky data (-A)** (Сбрасывать ненужные ("липкие") данные) — удалять информацию о тегах или ветках, которые "приклеились" к файлу;
- **Prune empty directories (-P)** (Сокращать пустые директории);

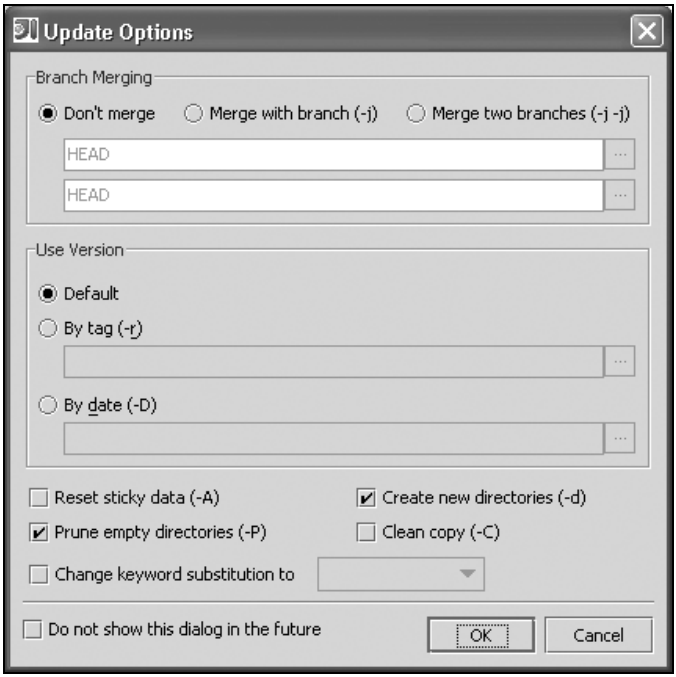


Рис. 12.32. Диалог настроек обновления проекта

- **Change keyword substitution to** (Изменить подстановку ключевого слово на):
  - binary (-kb);
  - compression (-kk);
  - expansion (-kkv);
  - expansion locker (-kkvl);
  - replacement (-kv);
  - no substitution (-ko);
- **Create new directories (-d)** (Создавать новые директории);
- **Clean copy (-C)** ("Чистое" копирование);
- **Do not show this dialog in the future** (Не показывать этот диалог в будущем).

Если во время обновления обнаружится, что файлы проекта или модулей изменились, IntelliJ IDEA предложит перезагрузить проект (рис. 12.33).

Результат обновления проекта будет выведен в инструментальном окне CVS (рис. 12.34).

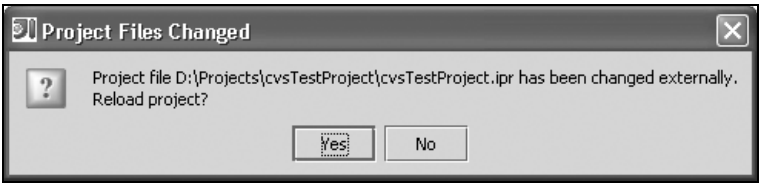


Рис. 12.33. Вопрос о перезагрузке проекта при обнаруженных изменениях в файле проекта

Рис. 12.34. Результат обновления проекта

12.3.4.4. *Check Project Status (Проверить статус проекта)*

Данная команда получает информацию из CVS о файлах проекта и отображает ее в инструментальном окне CVS.

Эту команду удобно использовать перед тем, как выкладывать в CVS большое количество изменений. Если над проектом трудится множество человек, может оказаться, что некоторые ваши изменения конфликтуют с изменениями в CVS. С помощью этой команды можно просмотреть такие конфликтные изменения до того, как выкладывать их в CVS и, соответственно, устранить.

12.3.4.5. *Global Settings (Глобальные настройки)*

Данная команда открывает диалог настроек CVS, актуальных для всех проектов в IntelliJ IDEA (*см. раздел 12.3.1.2*).

12.3.4.6. *Configure CVS Roots (Настроить репозитории CVS)*

Данная команда открывает диалог настройки конфигураций репозиториев CVS (*см. раздел 12.3.1.3*).

12.3.4.7. *Commit (Подтвердить)*

Подтвердить изменения в указанном файле, файлах директории или модуле. Данная команда открывает диалог, аналогичный тому, что используется для подтверждения изменений во всем проекте (*см. раздел 12.3.4.2*).

12.3.4.8. *Rollback Local Changes (Откатить локальные изменения)*

Данная команда позволяет откатить локальные изменения, сделанные в указанном файле, файлах директории или модуле. Перед исполнением данная команда отображает диалог с подтверждением (рис. 12.35).

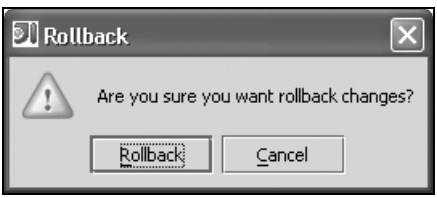


Рис. 12.35. Подтверждение отката локальных изменений

12.3.4.9. *Update (Обновить)*

Данная команда выполняет обновление выбранного локального файла, файлов в директории или модуле. Работа данной команды идентична работе команды **Update Project** (Обновить проект) (*см. раздел 12.3.4.3*).

12.3.4.10. *Check Out (Загрузить)*

Данная команда позволяет загрузить файлы из CVS. Ее можно вызвать для конкретного файла, группы файлов или директорий, для модуля. Если среди указанных для загрузки файлов имеются локально измененные, перед выполнением данная команда запросит подтверждение, поскольку загрузка измененных файлов приведет к потере всех изменений.

Данная команда открывает диалог, где можно выбрать версию файла, которую вы хотите загрузить (рис. 12.36).

Для загрузки текущей версии следует выбрать пункт **HEAD revision** (Текущая версия). Также можно указать тег или дату версии файла для загрузки.

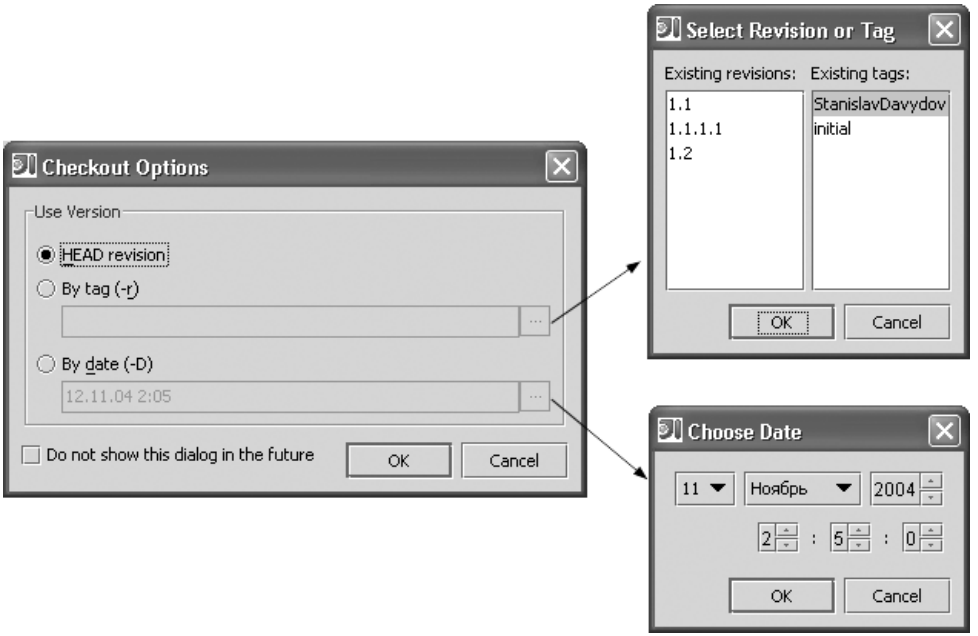


Рис. 12.36. Диалог загрузки файла из CVS

12.3.4.11. *Merge (Объединить)*

Если какой-либо файл имеет и локальные изменения, и изменения в CVS, для него можно выполнить команду **Merge** (Объединить), чтобы объединить эти изменения.

К сожалению, такое объединение возможно не всегда, поскольку часто изменения могут затрагивать одну и ту же строку. С объединением таких изменений CVS автоматически справиться не может.

Прежде для объединения таких файлов приходилось поработать руками в редакторе. В IntelliJ IDEA это не обязательно, поскольку в ней имеется встроенная функция визуального объединения файлов.

Данная команда открывает для таких файлов диалог, похожий на диалог сравнения, но в диалоге объединения имеется не две, а три панели сравнения (рис. 12.37).

На левой панели представлено содержимое локального файла, на правой панели — файла из CVS, а посередине — результат их объединения.

С помощью команд, расположенных в правом поле левой панели и левом поле правой панели, можно перемещать в центральную панель части содержимого из локальной версии файла и из CVS.

Текст на центральной панели также можно редактировать (работают все команды редактора (*см. раздел 3.6.4*)).

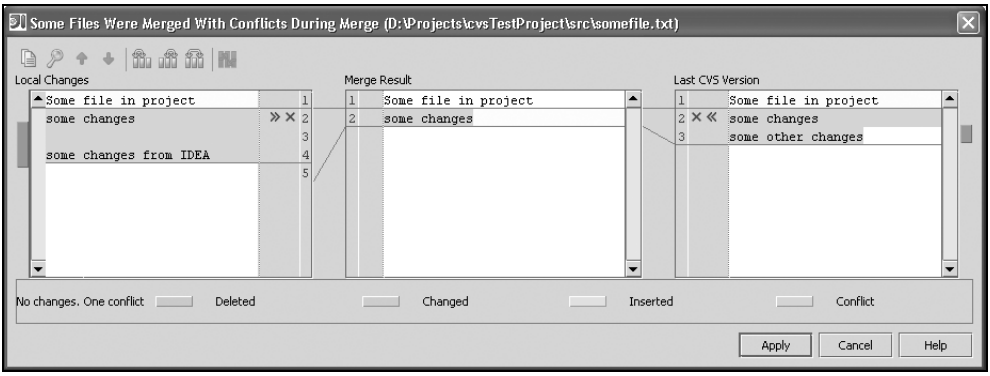


Рис. 12.37. Объединение версий файла

Дополнительно к командам окна сравнения (*см. раздел 12.1*) на панели инструментов имеются следующие команды:

- **Compare Merge result and Local Changes** (Сравнить результат объединения и локальные изменения) — команда открывает окно сравнения для текста, полученного в результате объединения, и локальных изменений;
- **Compare Merge result and Last CVS Version** (Сравнить результат объединения и последнюю версию из CVS) — команда открывает окно сравнения для текста из второй и третьей панелей;
- **Compare Local Changes and Last CVS Version** (Сравнить локальные изменения и последнюю версию из CVS) — команда открывает окно сравнения для текста из первой и второй панелей;
- **Apply all non-conflict changes** (Применить все неконфликтующие изменения).

12.3.4.12. *Add (Добавить)*

Команда добавляет в CVS файл, группу файлов или директорий. Данная команда открывает диалог, в котором перечислены все добавляемые файлы и директории, а также некоторые их настройки (рис. 12.38).

В верхней части диалога имеется панель инструментов с двумя командами:

- **Select All** (Выделить все) — включить в добавление все файлы;
- **Unselect All** (Исключить все) — исключить из добавления все файлы.

Для каждого файла в отдельности также можно задать подстановку ключевого слова (**Keyword Substitution**).

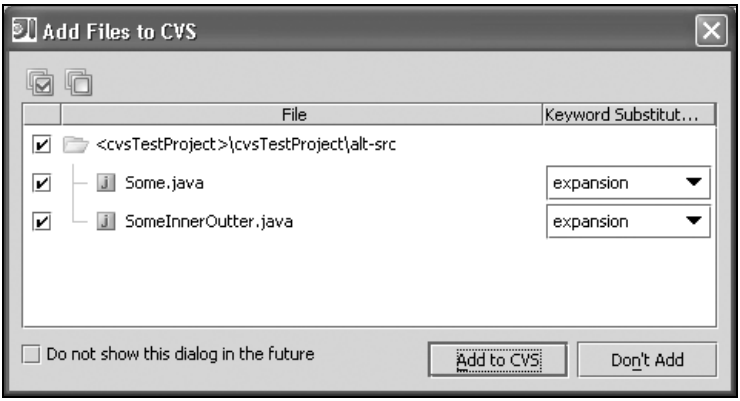


Рис. 12.38. Добавление файлов в CVS

12.3.4.13. *Undo 'Add' (Отменить добавление)*

Отменить добавление файлов (*см. раздел 12.3.4.12*).

12.3.4.14. *Ignore (Игнорировать)*

Включить файл, группу файлов или директорий в число тех, что будут игнорироваться при совершении операций CVS. Отменить игнорирование можно в диалоге подтверждения изменений (*см. раздел 12.3.4.2*).

12.3.4.15. *Compare with The Same Version (Сравнить с соответствующей версией)*

Данная команда открывает окно сравнения локальной версии файла с той же самой версией файла из CVS.

12.3.4.16. *Compare with Latest Repository Version (Сравнить с последней версией из репозитория)*

Данная команда открывает окно сравнения локальной версии файла с файлом из CVS самой последней версии.

12.3.4.17. *Compare with Selected Version (Сравнить с указанной версией)*

Данная команда открывает окно сравнения локальной версии файла с файлом из CVS с указанной версией. Данная команда позволяет указать версию файла с помощью диалога, аналогичному тому, что представлен на рис. 12.36.

12.3.4.18. *Check Status (Проверить статус)*

Команда проверяет статус выделенного файла, группы файлов, директорий или модуля. Работает аналогично команде **Check Project Status** (Проверить статус проекта) (*см. раздел 12.3.4.4*).

12.3.4.19. *Show History (Показать историю)*

Данная команда отображает историю изменений файла в инструментальном окне **CVS** (рис. 12.39).

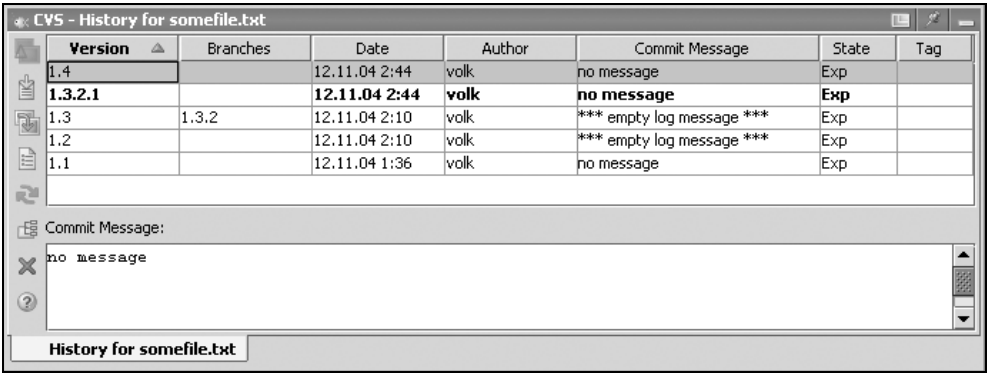


Рис. 12.39. История изменений файла

История изменений файла отображается в виде таблицы со следующими колонками:

- **Version** (Версия);
- **Branches** (Ветки);
- **Data** (Дата);
- **Author** (Автор);
- **Commit Message** (Сообщение при подтверждении изменений);
- **State** (Статус);
- **Tag** (Тег).

Если данный файл имеет изменения не только в основной ветке, но и в дополнительных, то есть возможность просмотреть эти изменения в виде иерархии веток (рис. 12.40). Для переключения в такой режим просмотра служит кнопка на панели инструментов **Show as Tree** (Показывать как дерево).

Version	Branches	Date	Author	Commit Message	State	Tag
1.1		12.11.04 1:36	volk	no message	Exp	
1.2		12.11.04 2:10	volk	*** empty log m...	Exp	
1.3	1.3.2	12.11.04 2:10	volk	*** empty log m...	Exp	
1.3.2.1		**12.11.04 2:44**	**volk**	**no message**	**Exp**	
1.4		12.11.04 2:44	volk	no message	Exp	

Рис. 12.40. Отображение версий в виде дерева

В левой части инструментального окна имеется панель инструментов со следующими командами:

- **Compare** (Сравнить) — если в таблице выделить две версии файла, для них можно выполнить сравнение изменений;
- **Compare with Local** (Сравнить с текущей версией) — команда позволяет сравнить выделенную версию файла в таблице версий с текущей локальной версией;
- **Get** (Получить) — команда загружает версию в локальный файл;
- **Annotate** (Аннотация) — команда отображает аннотацию для указанной версии файла (*см. раздел 12.3.4.21*);
- **Refresh** (Обновить) — команда обновляет информацию о версиях файла;
- **Show as Tree** (Показывать в виде дерева) — переключение режима просмотра таблицы;
- **Close** (Закрыть) — команда закрывает окно просмотра истории изменений;
- **Help** (Помощь) — команда открывает окно справочной системы.

В нижней части инструментального окна отображается сообщение, которое было задано выбранной версии файла при подтверждении (**Commit Message**).

12.3.4.20. *Show History for Selection (Показать историю изменений для выделения)*

Команда отображает историю изменений для выделенной части файла (рис. 12.41).

Данная команда отображает диалог, похожий на диалог сравнения файлов (*см. раздел 12.2.1*), в верхней части которого содержатся две панели с разными версиями выделенного содержимого, а в нижней части содержатся записи о версиях, в которых присутствует выделенный фрагмент. Информация об истории изменений аналогична той, что отображается командой **Show History** (Показать историю изменений) для файла (*см. раздел 12.3.4.19*).

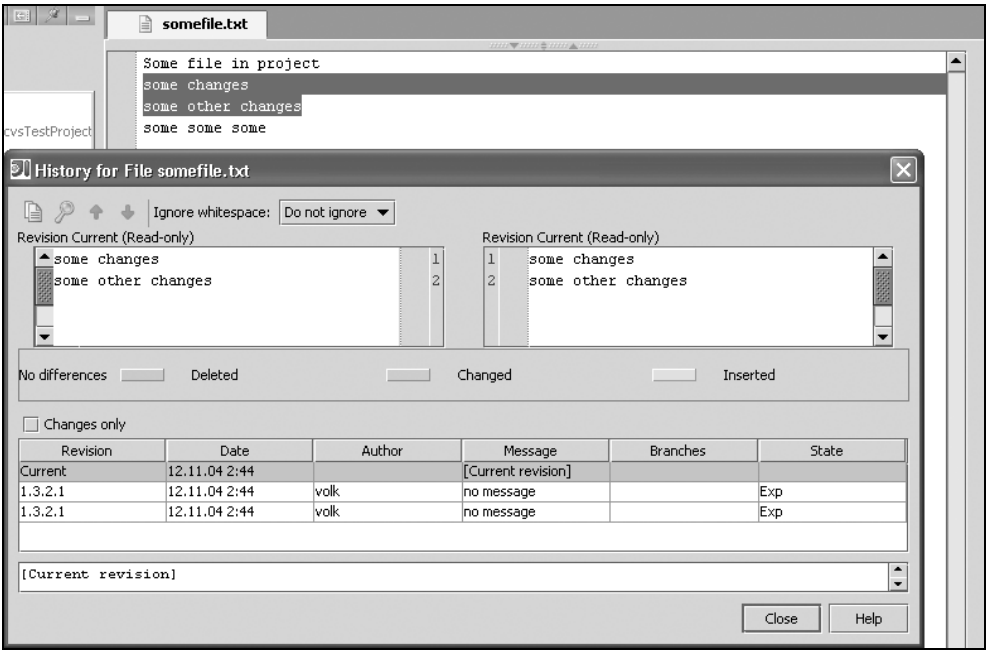


Рис. 12.41. История изменений выделенной части текста

12.3.4.21. *Annotate (Аннотировать)*

Данная команда выводит в левом поле редактора информацию о том, кто и когда модифицировал ту или иную строку (рис. 12.42).

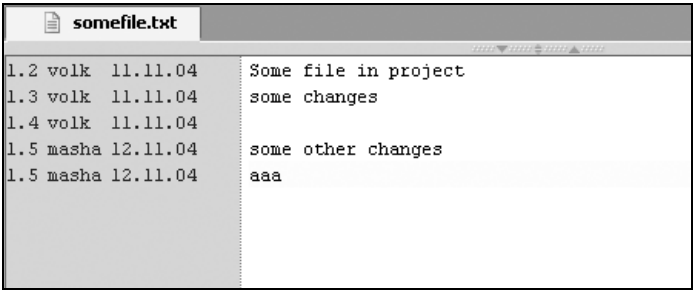


Рис. 12.42. Аннотации к строкам файла

В аннотации указывается версия, в которой была изменена данная строка, автор изменения, а также дата, когда было произведено изменение.

Чтобы закрыть аннотации, нужно щелкнуть правой кнопкой мыши на панели с аннотациями и в открывшемся контекстном меню выбрать единственную команду **Close Annotations** (Закрыть аннотации).

12.3.4.22. *Create Branch (Создать ветку)*

Данная команда позволяет создать новую ветку. При ее запуске открывается диалог, в котором можно задать название ветки, а также ее дополнительные свойства (рис. 12.43).

Рис. 12.43. Создание ветки

В диалоге имеются следующие параметры и опции:

- **Branch name** (Имя ветки);
- **Override existing (-F)** (Перезаписывать существующие) — опция указывает перезаписывать существующие теги с таким именем;
- **Switch to this branch** (Переключиться на эту ветку) — опция позволяет автоматически переключить на данную ветку файл, группу файлов или директорий, для которых была выполнена команда.

12.3.4.23. *Create Tag (Создать тег)*

Данная команда создает тег с указанным именем и свойствами (рис. 12.44).

Рис. 12.44. Создание тега

12.3.4.24. *Delete Tag (Удалить тег)*

Команда позволяет удалить тег (рис. 12.45).

Чтобы удалить тег, необходимо задать его имя. С помощью кнопки можно отобразить список доступных тегов и выбрать один из них.

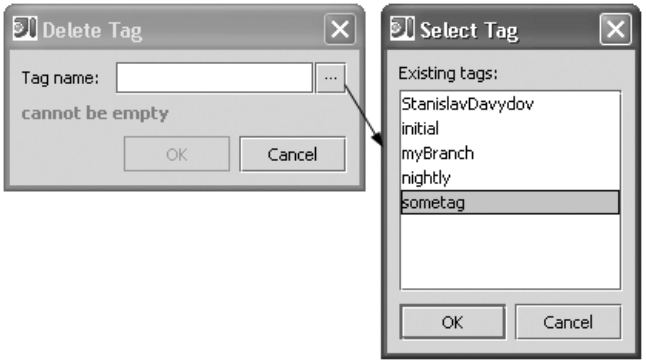


Рис. 12.45. Удаление тега

12.3.4.25. *Edit and Watch (Редактировать и наблюдать)*

Данный раздел меню содержит команды, облегчающие групповую работу с файлами в CVS.

Edit (Редактировать)

Пометить файл, группу файлов или директорий, как редактируемую данным пользователем (рис. 12.46).

Рис. 12.46. Дополнительные свойства команды **Edit**

С помощью диалога, представленного на рис. 12.46, можно задать следующие опции команды **Edit** (Редактировать):

- **Reserved edit (-c)** (Зарезервировано для редактирования);
- **Do not show this dialog in the future** (Больше не показывать этот диалог).

Unedit (Отменить редактировать)

Отменить редактирование для указанного файла, группы файлов или директорий.

Show Editors (Показать редакторы)

Данная команда отображает список редакторов выбранного файла, группы файлов или директорий (рис. 12.47).

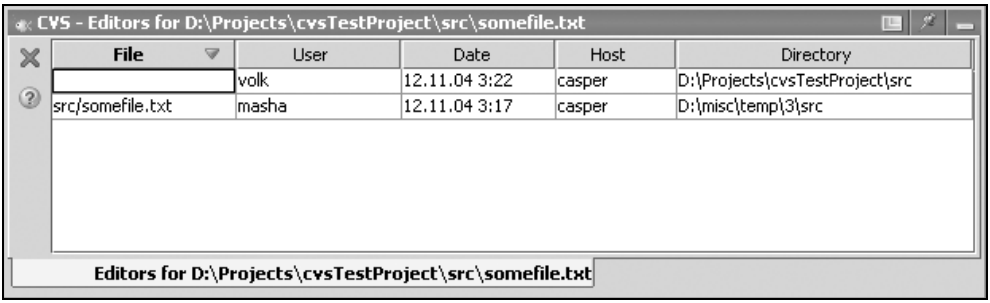


Рис. 12.47. Просмотр редакторов

Информация о редакторах отображается в инструментальном окне **CVS** в виде таблицы со следующими колонками:

- **File** (Файл) — редактируемый файл;
- **User** (Пользователь) — имя пользователя, который редактирует файл;
- **Date** (Дата) — дата начала редактирования;
- **Host** (Хост) — хост, на котором происходит редактирование;
- **Directory** (Директория) — директория, в которой находится редактируемый файл.

Watch On (Включить наблюдение)

Включить наблюдение за указанным файлом, группой файлов или директорий.

Watch Off (Отключить наблюдение)

Отключить наблюдение за указанным файлом, группой файлов или директорий.

Add Watch (Добавить наблюдение)

Добавить наблюдение за указанным файлом, группой файлов или директорий. Данная команда открывает диалог, в котором можно указать, за какими именно операциями вы хотите наблюдать (рис. 12.48).

Рис. 12.48. Добавление наблюдения

Имеются следующие возможности для наблюдения (**Watching actions**):

- **Edit** (Наблюдать за началом редактирования);
- **Unedit** (Наблюдать за прекращением редактирования);

- **Commit** (Наблюдать за подтверждением изменений);
- **All** (Наблюдать за всеми перечисленными действиями).

Remove Watch (Снять наблюдение)

Снять наблюдение за указанным файлом, группой файлов или директорий.

Show Watchers (Показать наблюдателей)

Показать наблюдателей за указанным файлом, группой файлов или директорий. Данная команда открывает инструментальное окно **CVS** с информацией о наблюдателях (рис. 12.49).

Рис. 12.49. Просмотр информации о наблюдателях

Список наблюдателей выводится в виде таблицы со следующими столбцами:

- **File** (Файл) — файл, за которым наблюдает наблюдатель;
- **User** (Пользователь) — имя пользователя-наблюдателя;
- **Actions** (Действия) — действия, за которыми наблюдает наблюдатель.

12.4. SourceSafe

SourceSafe — система контроля версий фирмы Microsoft. Рассмотрим настройки проекта для работы с данной системой контроля версий.

12.4.1. Настройки

Настройка поддержки SourceSafe для проекта в IntelliJ IDEA предполагает, что у вас уже имеется сконфигурированный проект в SourceSafe, хотя бы в минимальном объеме (рис. 12.50).

Настройка SourceSafe производится с помощью диалога настройки системы контроля версий для проекта (*см. раздел 4.2.3*), представленного на рис. 12.51.

В данном диалоге имеются две закладки с опциями:

- **General** (Основные настройки):
  - **Path to VSS client (ss.exe)** (Путь к клиенту Visual SourceSafe);
  - **Path to VSS configuration file (srcsafe.ini)** (Путь к файлу конфигурации Visual SourceSafe);

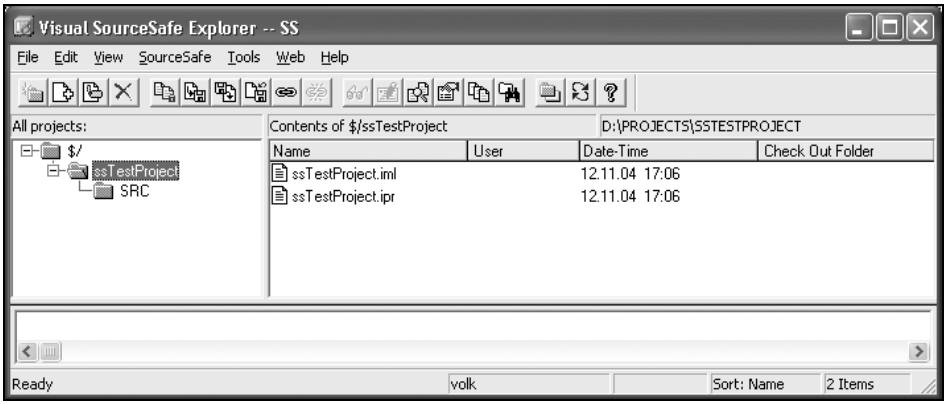


Рис. 12.50. Проект в SourceSafe

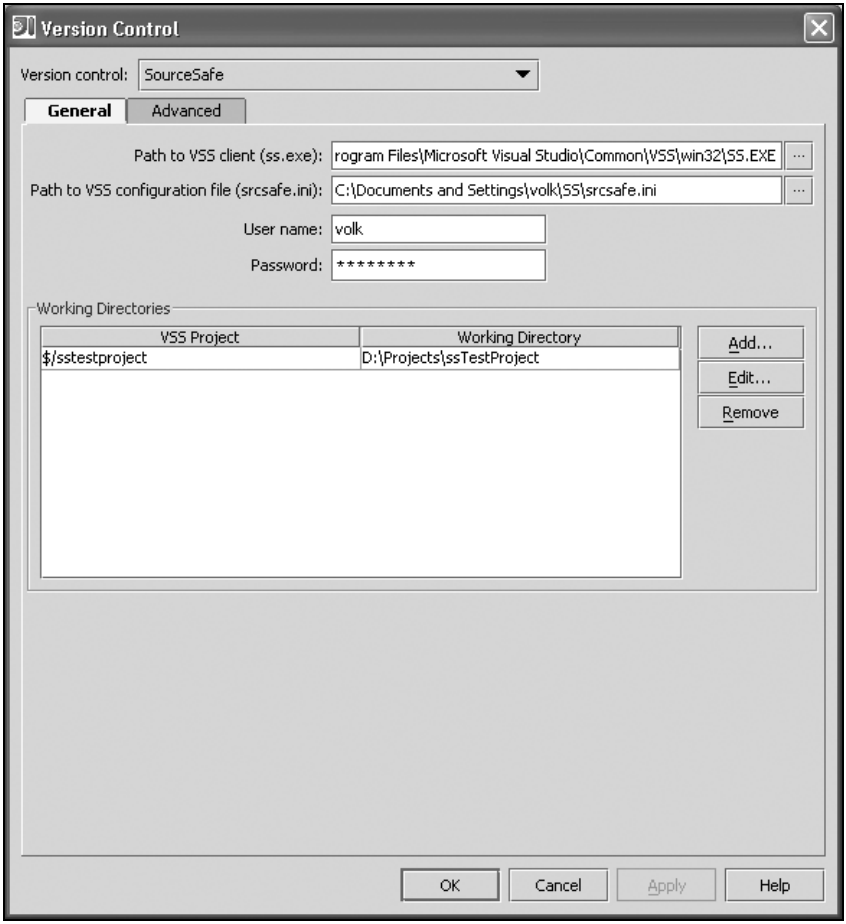


Рис. 12.51. Настройка SourceSafe для проекта

- **User name** (Имя пользователя);
- **Password** (Пароль);
- **Working Directories** (Рабочие директории) — полный перечень рабочих директорий, используемых проектом. Директории можно добавлять (с помощью кнопки **Add** (Добавить)), редактировать (с помощью кнопки **Edit** (Редактировать)) (см. рис. 12.52) и удалять (с помощью кнопки **Delete** (Удалить));

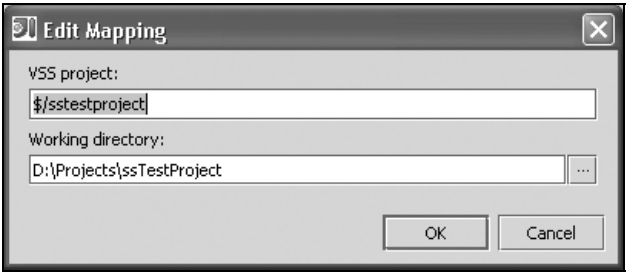


Рис. 12.52. Редактирование рабочей директории

❒ **Advanced** (Дополнительные настройки):

- **Command dialogs** (Диалоги команд) — данная группа опций задает команды системы контроля версий, при выполнении которых будет открываться диалог с дополнительными параметрами (**Display option dialog when this commands are invoked**):
  - ◊ **Check Out** (Загрузка) — загрузка файлов из системы контроля версий;
  - ◊ **Check In** (Запись) — запись изменений в систему контроля версий;
  - ◊ **Add File** (Добавление файла);
  - ◊ **Undo Check Out** (Отмена загрузки);
  - ◊ **Compare with SourceSafe Version** (Сравнение с версией в SourceSafe);
  - ◊ **Get Latest Version** (Получение последней версии);
- **Comment Options** (Настройки комментариев):
  - ◊ **Reuse last comment** (Использовать последний комментарий);
  - ◊ **Put focus into comment** (Помещать курсор в поле ввода комментария).

12.4.2. Визуальное выделение файлов

В зависимости от того, какой статус имеет тот или иной файл в проекте относительно системы контроля версий, он может быть выделен разными цветами:

❒ черный (файл имеет текущую версию);

- зеленый (новый файл еще не добавлен в систему контроля версий);
- синий (файл изменен локально);
- серый (удаленный файл).

12.4.3. Команды

Команды для работы с SourceSafe собраны в разделе **SourceSafe** главного меню. Данный раздел появляется в главном меню, только если текущий проект настроен использовать SourceSafe в качестве системы контроля версий.

12.4.3.1. *Check In Project (Подтвердить изменения в проекте)*

Данная команда выполняет подтверждение изменений и добавление файлов в систему контроля версий для всех файлов проекта. Команда открывает диалог, представленный на рис. 12.53, где отображены все изменения в проекте, которые будут записаны в систему контроля версий.

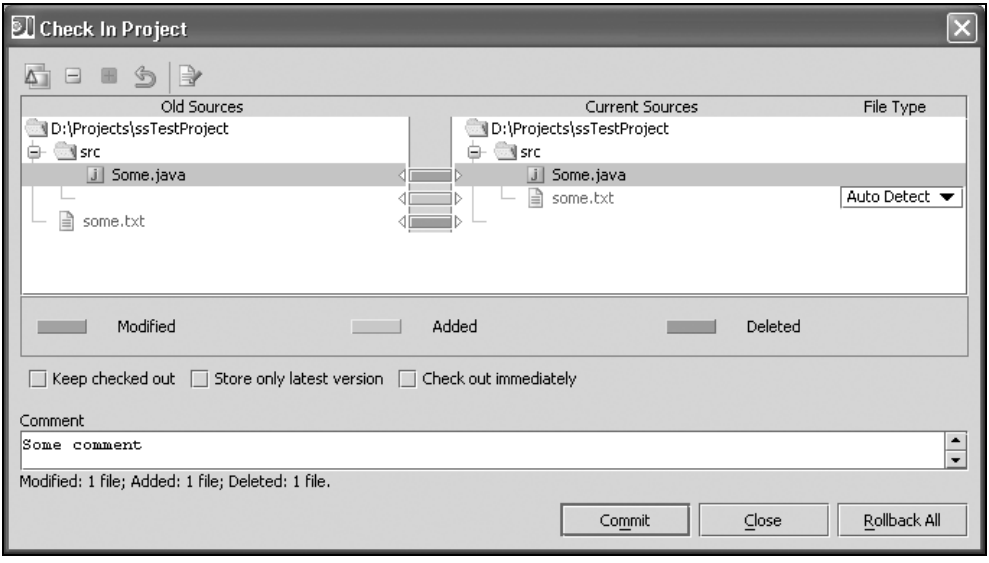


Рис. 12.53. Добавление изменений в систему контроля версий

В верхней части диалога имеется панель инструментов, центральную часть занимают две панели, на которых отображаются изменения, произошедшие в проекте. В нижней части диалога имеются дополнительные опции данной операции.

На панели инструментов представлены следующие команды:

- **Show Changes** (Показать изменения) — показать изменения в выделенном файле;

- **Exclude from Commit** (Исключить из записи изменений) — команда позволяет исключить файл или директорию со всеми файлами в ней из записи в систему контроля версий;
- **Include into Commit** (Включить в запись изменений) — команда позволяет включить обратно файл или директории в запись изменений;
- **Rollback** (Откатить) — откатить локальные изменения;
- **Edit Source** (Редактировать исходный текст) — команда открывает в редакторе указанный файл.

В диалоге представлены следующие дополнительные опции и параметры:

- **Keep checked out** (Сохранять загруженными) — опция позволяет оставить файлы в режиме загруженных (т. е. доступных для редактирования);
- **Store only latest version** (Сохранять только последнее изменение);
- **Check out immediately** (Загрузить сразу после подтверждения изменений);
- **Comment** (Комментарий) — комментарий к изменениям.

В правой нижней части диалога имеются три управляющие кнопки:

- **Commit** (Подтвердить) — подтвердить изменения;
- **Close** (Закрыть) — закрыть диалог без подтверждения;
- **Rollback All** (Откатить все) — откатить все сделанные изменения.

12.4.3.2. *Open SourceSafe Explorer (Открыть SourceSafe Explorer)*

Команда загружает программу SourceSafe Explorer (см. рис. 12.50).

12.4.3.3. *Mark Project as Current (Пометить проект как текущий)*

Команда помечает проект как текущий.

12.4.3.4. *Check Out (Загрузить)*

Команда загрузки файлов из SourceSafe. Перед загрузкой данная команда открывает диалог, представленный на рис. 12.54.

В данном диалоге можно дополнительно ввести комментарий (**Comment**), а также две опции:

- **Don't get local copy** (Не получать локальную копию);
- **Skip this dialog** (Больше не выводить этот диалог).

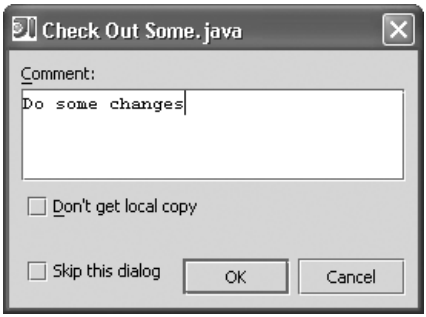


Рис. 12.54. Диалог загрузки файлов

12.4.3.5. *Check In (Подтвердить изменения)*

Команда подтверждает локальные изменения и записывает их в SourceSafe. Команда перед записью открывает диалог, представленный на рис. 12.55, в котором можно задать следующие параметры и опции:

- **Comment** (Комментарий к изменениям);
- **Keep checked out** (Оставить файлы загруженными);
- **Skip this dialog** (Больше не выводить этот диалог).

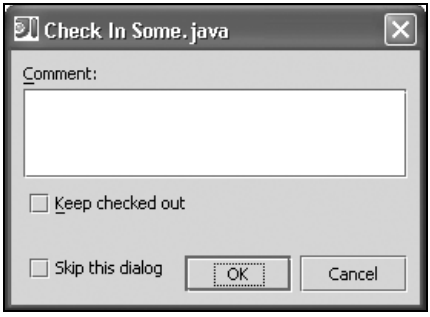


Рис. 12.55. Диалог подтверждения изменений

12.4.3.6. *Add File (Добавить файл)*

Команда добавляет новый файл в SourceSafe. Перед добавлением команда открывает диалог, представленный на рис. 12.56.

В данном диалоге можно задать дополнительные опции и параметры:

- **Comment** (Комментарий к файлу);
- **Advanced Options** (Расширенные опции):
  - **Store only latest version** (Сохранять только последнюю версию);
  - **Check out immediately** (Загрузить незамедлительно);

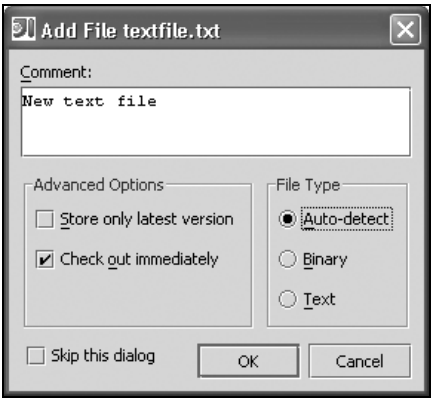


Рис. 12.56. Диалог добавления файла

- **File Type** (Тип файла):
  - **Auto-detect** (Распознавать автоматически);
  - **Binary** (Бинарный файл);
  - **Text** (Текстовый файл);
- **Skip this dialog** (Больше не показывать этот диалог).

12.4.3.7. *Undo Check Out (Отменить загрузку)*

Команда отмены загрузки файла (*см. раздел 12.4.3.4*). Перед отменой данная команда открывает диалог, представленный на рис. 12.57.

Рис. 12.57. Диалог отмены загрузки файла

В данном диалоге можно задать дополнительные опции:

- **Replace local copy** (Заменить локальную копию для измененных файлов):
  - **Ask** (Запросить подтверждение);
  - **Leave** (Не заменять);
  - **Replace** (Заменить);
- **Make writable** (Пометить файл как доступный для записи);
- **Skip this dialog** (Больше не выводить этот диалог).

12.4.3.8. *Compare with SourceSafe Version (Сравнить с версией из SourceSafe)*

Данная команда открывает окно сравнения локальной версии файла и версии из SourceSafe. Про сравнение файлов *см. раздел 12.1.1.*

12.4.3.9. *Get Latest Version (Получить последнюю версию)*

Данная команда получает из SourceSafe последнюю версию указанного файла. Перед выполнением данная команда отображает диалог, аналогичный тому, который выводит команда **Undo Check Out** (Отменить загрузку) (*см. раздел 12.4.3.7*).

12.5. StarTeam

StarTeam — система контроля версий фирмы StarGate, которая пару лет назад была куплена фирмой Borland.

12.5.1. Настройки

Общий вид диалога настроек интеграции проекта с системой контроля версий StarTeam представлен на рис. 12.58.

12.5.2. Визуальное выделение файлов

В зависимости от статуса того или иного файла в проекте относительно системы контроля версий, файл может быть выделен разными цветами:

- черный (файл имеет текущую версию);
- зеленый (новый файл еще не добавлен в систему контроля версий);
- синий (файл изменен локально);
- серый (удаленный файл).

12.5.3. Команды

Команды для работы со StarTeam собраны в разделе **StarTeam** главного меню. Данный раздел появляется в главном меню, только если текущий проект имеет настройки использовать StarTeam в качестве системы контроля версий.

12.5.3.1. *Mark Project as Current (Пометить проект как текущий)*

Команда помечает проект как текущий.

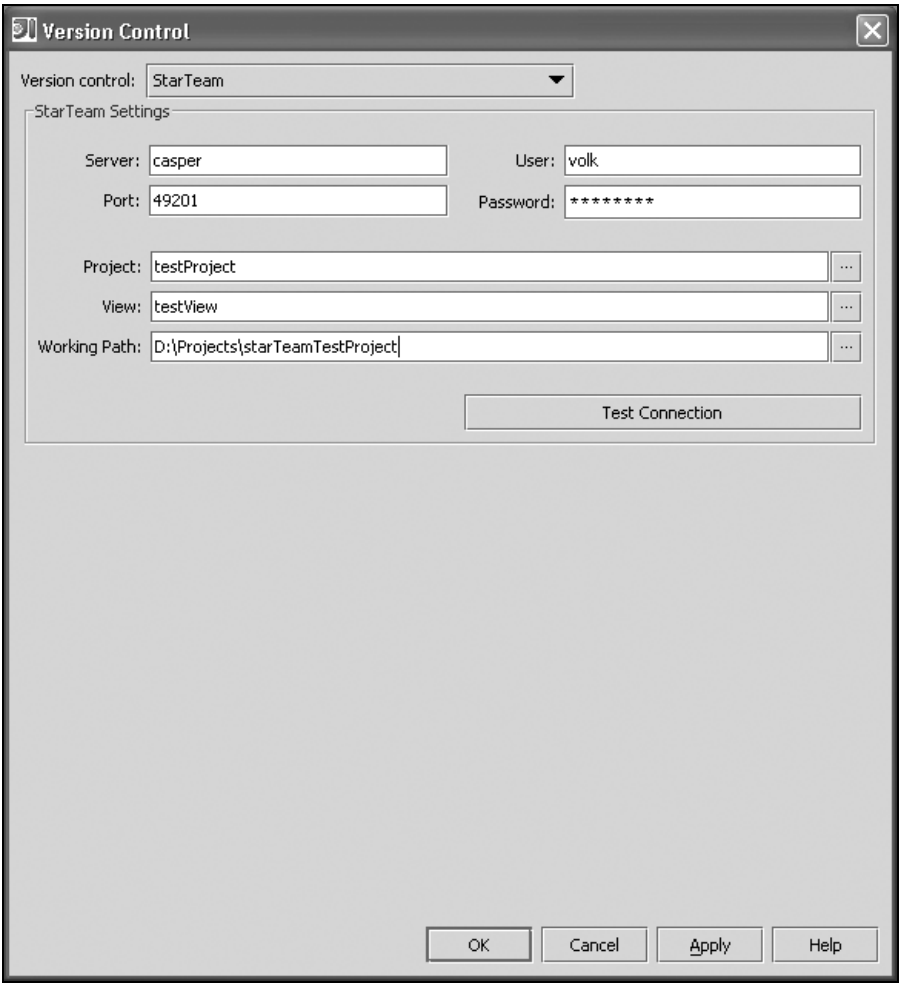


Рис. 12.58. Настройки интеграции со StarTeam

12.5.3.2. *Check In Project (Подтвердить изменения в проекте)*

Данная команда выполняет подтверждение изменений и добавление файлов в систему контроля версий для всех файлов проекта. Команда открывает диалог, представленный на рис. 12.59, в котором отображены все изменения в проекте, которые будут записаны в систему контроля версий.

В верхней части диалога расположена панель инструментов, центральную часть занимают две панели, в которых отображаются изменения, произошедшие в проекте. В нижней части диалога имеются дополнительные опции данной операции.

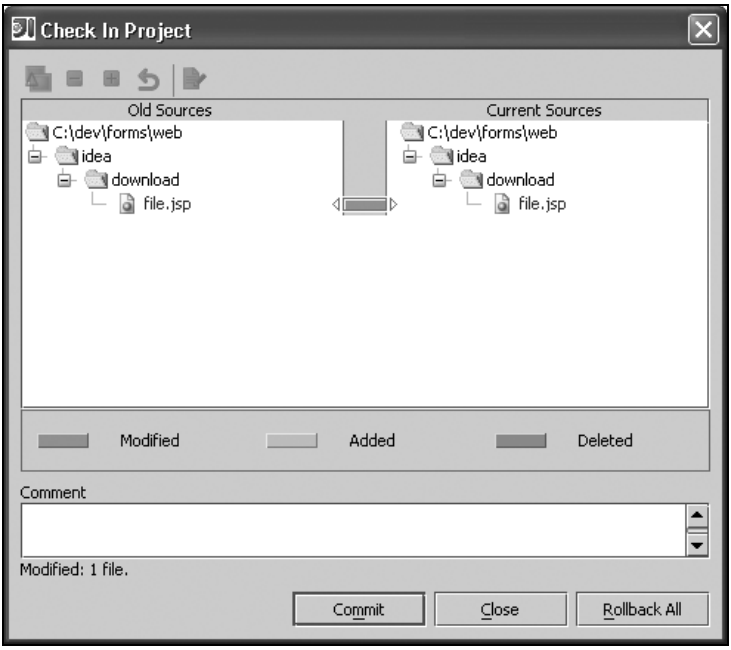


Рис. 12.59. Добавление изменений в систему контроля версий

На панели инструментов представлены следующие команды:

- **Show Changes** (Показать изменения) — показать изменения в выделенном файле;
- **Exclude from Commit** (Исключить из записи изменений) — команда позволяет исключить файл или директорию со всеми файлами в ней из записи в систему контроля версий;
- **Include into Commit** (Включить в запись изменений) — команда позволяет включить обратно файл или директории в запись изменений;
- **Rollback** (Откатить) — откатить локальные изменения;
- **Edit Source** (Редактировать исходный текст) — команда открывает в редакторе указанный файл.

12.5.3.3. *Add (Добавить)*

Команда добавления файла или директории в репозиторий StarTeam.

12.5.3.4. *Check Out (Загрузить)*

Команда выполняет загрузку файлов из репозитория StarTeam. Если некоторые из загружаемых файлов модифицированы локально, IntelliJ IDEA выведет предупреждающее сообщение о возможной потере данных.

12.5.3.5. *Lock (Заблокировать)*

Заблокировать возможность изменения файла, группы файлов или директории. Данная команда предоставляет исключительное право на редактирование указанного файла пользователю, вызвавшему эту команду.

12.5.3.6. *Unlock (Разблокировать)*

Команда, снимающая блокировку изменений, установленных предыдущей командой.

12.5.3.7. *Check In (Подтвердить изменения)*

Команда сохраняет изменения в файле, группе файлов или директорий в StarTeam. При выполнении данной команды имеется возможность указать комментарий к изменению (рис. 12.60).

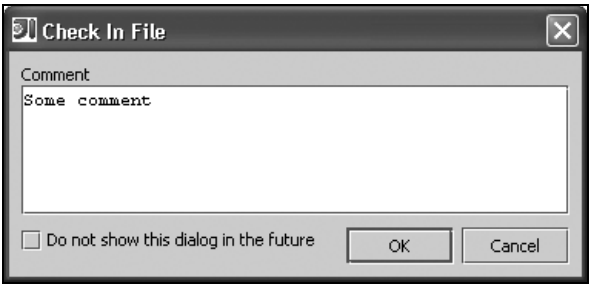


Рис. 12.60. Подтверждение изменений

12.5.3.8. *Show History (Показать историю изменений)*

Команда отображает список изменений, произведенных с указанным файлом (рис. 12.61).

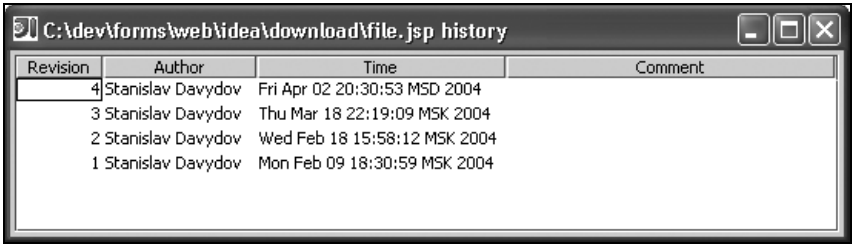


Рис. 12.61. История изменений файла

12.5.3.9. *Compare with StarTeam Version (Сравнить с последней версией в StartTeam)*

Данная команда открывает диалог сравнения для двух версий файла — локальной и последней версией из репозитория StarTeam. Подробнее о сравнении файлов *см. раздел 12.1.*

12.5.3.10. *Refresh (Обновить)*

Команда обновления информации из репозитория StarTeam.

12.5.3.11. *Reconnect (Переустановить связь)*

Переподключение к репозиторию StarTeam.

12.6. Дополнительные команды

Когда проект в IntelliJ IDEA имеет связь с системой контроля версий, стандартные возможности редактора файлов (*см. раздел 3.6*) расширяются дополнительной функцией перехода между локальными изменениями. В левом поле редактора появляются цветные метки, обозначающие места в файле, которые изменились по сравнению с версией в репозитории системы контроля версий (рис. 12.62). Цветовые метки в правом поле их дублируют.

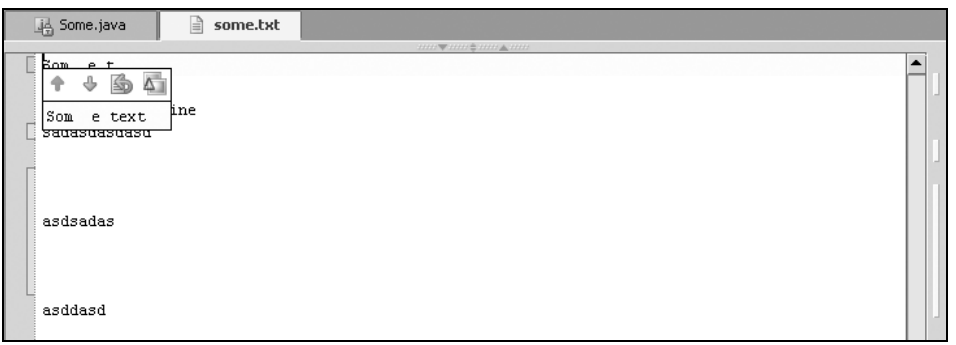
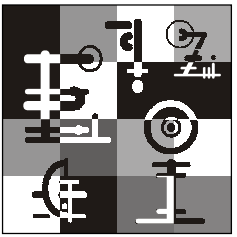


Рис. 12.62. Дополнительные функции в окне редактора

Метки имеют разные цвета, в зависимости от того, какое именно изменение было произведено с текстом (добавление, изменение или удаление).

При нажатии на метку в левом поле открывается всплывающее окно, в котором отображается оригинальный текст до изменений. В данном всплывающем окне имеется панель инструментов со следующими командами:

- **Go To Prev/Next Change** (Переход к предыдущему/следующему изменению);
- **Rollback** (Откатить) — откатить изменение;
- **Show Difference** (Показать отличия) — команда открывает окно сравнения оригинального текста и измененного (*см. раздел 12.1.2*).



Глава 13

Утилита Ant

Утилита сборки Ant является для проектов на Java аналогом утилиты make для проектов на C/C++. Данная утилита разрабатывается в рамках проекта Apache/Jakarta.

В состав IntelliJ IDEA включена Ant версии 1.6.1, поэтому, если данная версия вас устраивает, устанавливать Ant отдельно не требуется. Также поддерживаются другие версии Ant, которые можно настроить с помощью диалога настроек **Ant Build** (*см. раздел 13.3*).

В данной главе мы рассмотрим основные возможности IntelliJ IDEA по взаимодействию с утилитой Ant, а также поддержку редактирования файлов сборки Ant.

13.1. Создание и подключение файлов сборки

Если в вашем проекте уже имеется файл (или файлы) для сборки утилитой Ant, все что необходимо — это подключить их к проекту IntelliJ IDEA. Если такого файла еще нет, нужно его создать с помощью команды **Edit | New** (Редактировать | Создать). Обычно файлы сборки Ant называются build.xml (рис. 13.1).

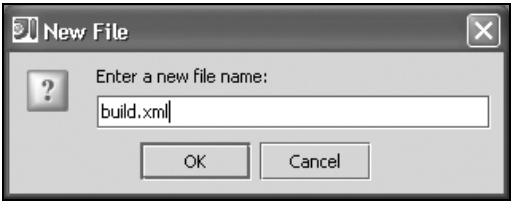


Рис. 13.1. Создание файла сборки Ant

После того как файл создан, его можно добавить в проект IntelliJ IDEA. Сделать это можно двумя способами:

- С помощью команды **Add as Ant Build File** (Добавить как файл сборки утилиты Ant) из меню раздела **Tools** (Утилиты). Данная команда появляется в меню **Tools** (Утилиты), только если текущим файлом является XML-файл.
- С помощью кнопки **Add** (Добавить) с панели инструментов инструментального окна **Ant Build** (Сборка Ant) (*см. раздел 3.4.8.10*).

Когда вы добавите файл, он появится в виде элемента списка в инструментальном окне **Ant Build** (Сборка Ant) (рис. 13.2).

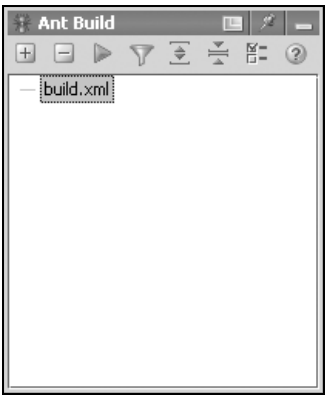


Рис. 13.2. Файл сборки добавлен в список

Так как в нашем файле build.xml нет никакого содержимого, в инструментальном окне больше ничего не отображается. Если вы добавляли в проект файл, в котором уже имеются цели для сборки, они бы высветились в списке.

Прежде чем создать цель для сборки, нужно создать тег проекта (рис. 13.3).

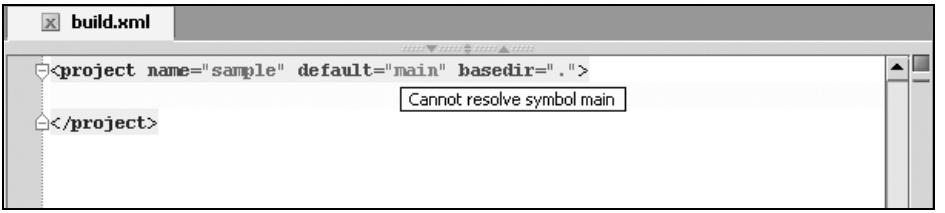


Рис. 13.3. Создание тега проекта

IntelliJ IDEA подсвечивает значение атрибута `default` и сообщает, что не удается найти символ `main`. Атрибут `default` задает цель, выполняемую по умолчанию при сборке. Если подвести к этому атрибуту курсор, через неболь-

шой промежуток времени появится подсказка с предложением создать цель с указанным нами именем `main` (рис. 13.4).

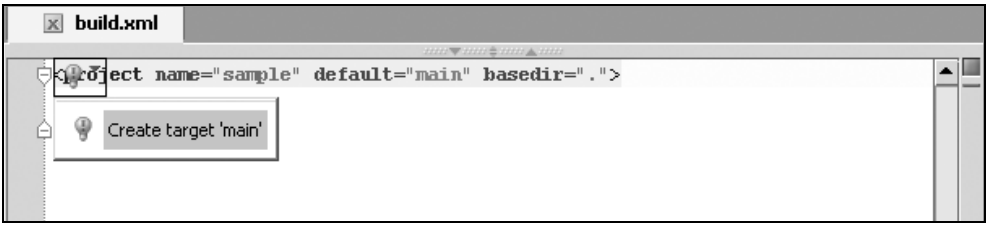


Рис. 13.4. Подсказка о создании цели `main`

Мы могли бы и сами создать такую цель, но позволим IntelliJ IDEA позаботиться об этом. В итоге мы получим файл сборки, представленный в листинге 13.1. Это самый простейший файл сборки, который пока ничего не делает.

Листинг 13.1

```
<project name="sample" default="main" basedir=".">
    <target name="main"/>
</project>
```

Если мы заглянем в инструментальное окно **Ant Build** (Сборка Ant), то увидим, что цель `main` добавилась в список (рис. 13.5).

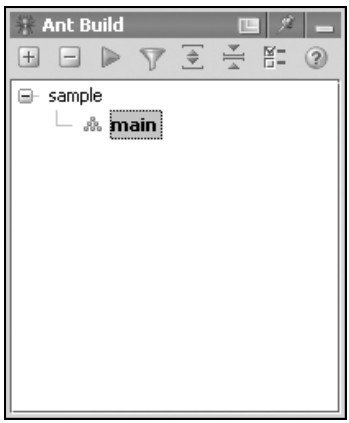


Рис. 13.5. Цель `main` добавилась в список

Цели в списке имеют значок из сложенных горкой ядер. По мере добавления целей в файл сборки они будут появляться в списке.

13.2. Редактирование файлов сборки

Так как файлы сборки Ant являются XML-файлами, для них применимы все функции редактирования XML-файлов в IntelliJ IDEA (*см. раздел 10.1*).

Кроме базовых возможностей, в XML-файлах Ant работают специальные возможности.

13.2.1. Список доступных команд и параметров

Список доступных команд при редактировании XML показан на рис. 13.6.

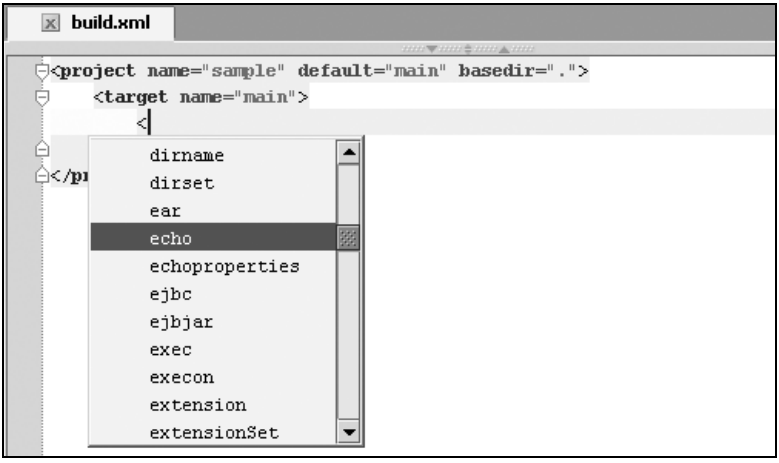


Рис. 13.6. Вывод списка доступных команд Ant при редактировании XML-файла

Вывод списка доступных параметров (атрибутов тега) команды Ant показан на рис. 13.7.

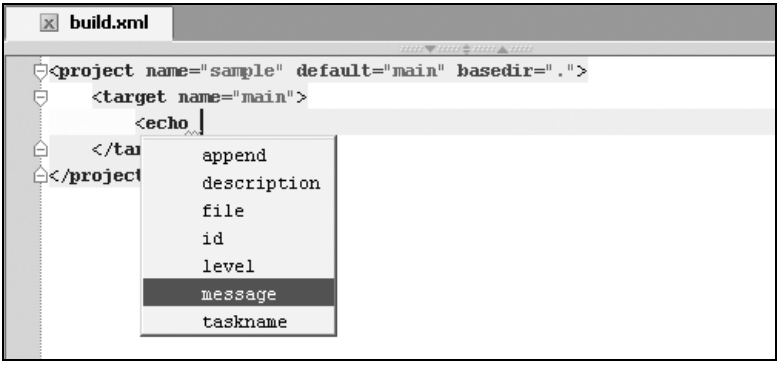


Рис. 13.7. Список доступных параметров команды

13.2.2. Вывод дополнительной информации

Для тегов команд Ant работают команды просмотра дополнительной информации о команде:

- Просмотр информации о параметрах (<Ctrl>+<P>) (рис. 13.8).

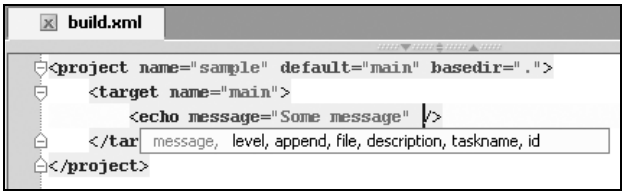


Рис. 13.8. Просмотр информации о параметрах команды (<Ctrl>+<P>)

Рис. 13.9. Просмотр комментариев для команд Ant

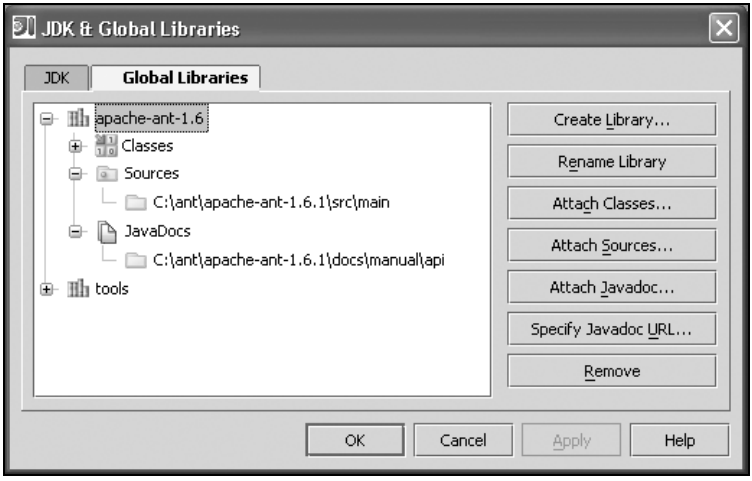


Рис. 13.10. Настройка библиотеки для Ant

- Просмотр комментариев JavaDoc для команд Ant (<Ctrl>+<Q>) (рис. 13.9). Данная функция работает, только если у вас настроена библиотека, содержащая JAR-файлы утилиты Ant, а также исходные файлы или комментарии JavaDoc (рис. 13.10).

Подробнее о настройке библиотек *см. раздел 4.1.8.2.*

13.2.3. Проверка ошибок

В файлах сборки Ant работает автоматическая проверка ошибок, например, проверка использования несуществующих свойств, команд или прочих конструкций, которые могут привести к ошибке (см. рис. 13.3, 13.11).

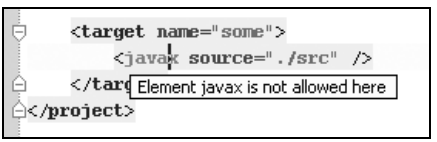


Рис. 13.11. Вывод информации об ошибках

13.2.4. Редактирование зависимостей

При редактировании целей Ant, которые зависят от других целей, IntelliJ IDEA может предложить варианты зависимостей (рис. 13.12).

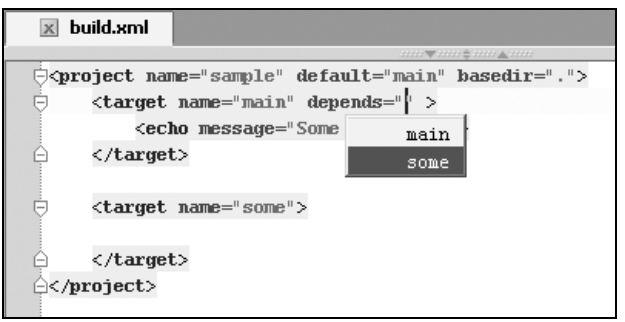


Рис. 13.12. Выбор зависимостей

Если в качестве зависимости ввести название цели, которой не существует в файле сборки, IntelliJ IDEA предложит создать такую цель.

Для целей также имеется возможность использовать рефакторинг переименования (рис. 13.13).

В случае такого рефакторинга будут корректно переименованы все вхождения имени данной цели, в том числе и в описании зависимостей других целей.

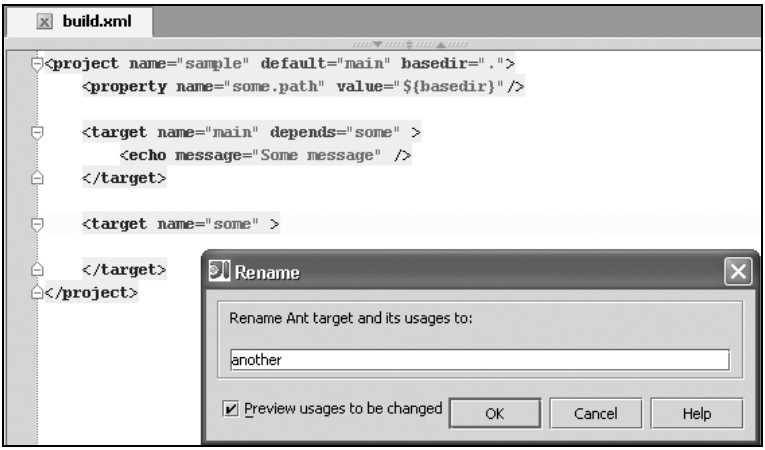


Рис. 13.13. Переименование целей

13.2.5. Использование свойств

В именах свойств (property), а также в вызовах целей Ant из других целей с помощью команды **Antcall** (Вызов Ant), работает команда перехода к определению (<Ctrl>+<B>) свойства или цели. Подробнее о команде перехода к определению *см. раздел 3.2.5.5.*

13.3. Настройка параметров

Настройка параметров производится с помощью диалога, представленного на рис. 13.14. Данный диалог можно открыть с помощью кнопки **Properties** (Свойства) с панели инструментов окна **Ant Build** (Сборка Ant).

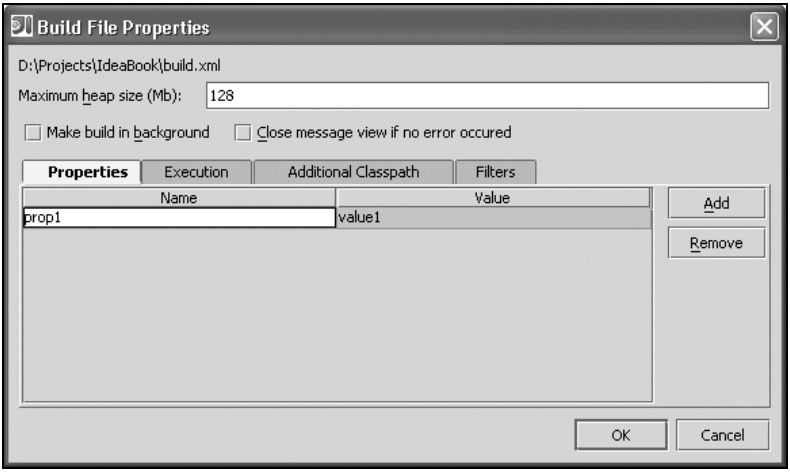


Рис. 13.14. Свойства сборки Ant

13.3.1. Общие свойства

Данный диалог содержит следующие опции:

- **Maximum heap size** (Максимальный размер памяти) — максимальный размер памяти в мегабайтах, выделяемый для запуска сборки данного файла Ant;
- **Make build in background** (Выполнять сборку в фоновом режиме);
- **Close message view if no error occurred** (Закрывать окно сообщений, если ошибок не произошло).

13.3.2. Параметры для передачи файлу сборки

Данные свойства представлены на закладке **Properties** (Параметры) (рис. 13.14).

С помощью данной закладки можно составить список свойств, которые будут переданы файлу сборки Ant. Свойства можно добавлять (с помощью кнопки **Add** (Добавить)) и удалять (с помощью кнопки **Remove** (Удалить)).

13.3.3. Выполнение

Свойства выполнения сборки представлены на закладке **Execution** (Выполнение) (рис. 13.15).

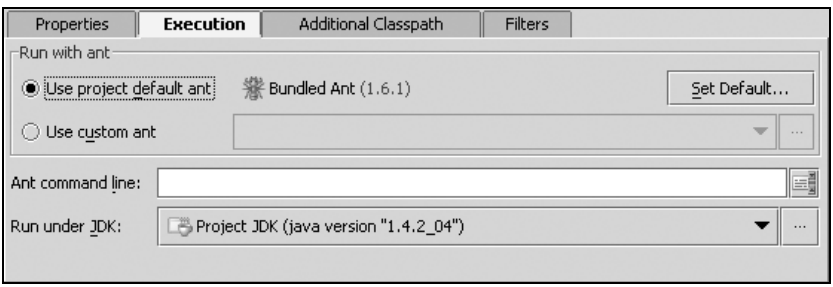


Рис. 13.15. Закладка настроек выполнения

На данной закладке представлены следующие опции:

- **Run with ant** (Запускать утилитой Ant):
  - **Use project default ant** (Использовать версию по умолчанию для проекта);
  - **Use custom ant** (Использовать другую версию);
- **Ant command line** (Дополнительные параметры командной строки для Ant);
- **Run under JDK** (Запускать с указанной версией JDK).

С помощью кнопки **Set Default** (Настроить версию Ant по умолчанию) можно открыть диалог настройки доступных версий Ant и выбрать ту версию,

которую вы хотите иметь в качестве версии по умолчанию для всего проекта. Данная кнопка открывает диалог, представленный на рис. 13.16.

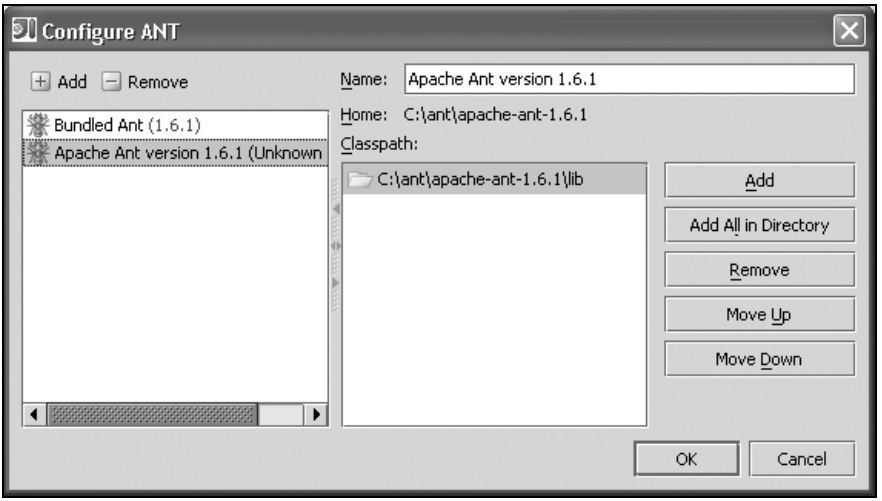


Рис. 13.16. Настройка версий Ant

С помощью кнопки **Add** (Добавить) с панели инструментов диалога можно добавить версию Ant, с помощью кнопки **Remove** (Удалить) — удалить. Версия Bundled Ant (1.6.1) не может быть удалена, поскольку она поставляется вместе с IntelliJ IDEA и встроена в систему.

При добавлении версии Ant имеется возможность настроить набор библиотек (JAR-файлов), которые будут использованы для запуска Ant. С помощью кнопок в правой части диалога можно:

- добавить библиотеку указанной версии Ant (кнопка **Add** (Добавить));
- добавить все файлы библиотек из указанной директории (кнопка **Add All in Directory** (Добавить все из директории));
- удалить указанную библиотеку (кнопка **Remove** (Удалить));
- изменить порядок библиотек в списке (кнопки **Move Up/Down** (Переместить вверх/вниз)).

13.3.4. Дополнительные библиотеки

С помощью списка на закладке **Additional Classpath** (Дополнительные библиотеки) можно добавить к имеющимся библиотекам утилиты Ant дополнительные (рис. 13.17).

Если в вашем файле сборки Ant используются дополнительные команды, которых нет в стандартной поставке Ant, с помощью данной закладки можно

добавить библиотеки, в которых находится имплементация этих дополнительных команд.

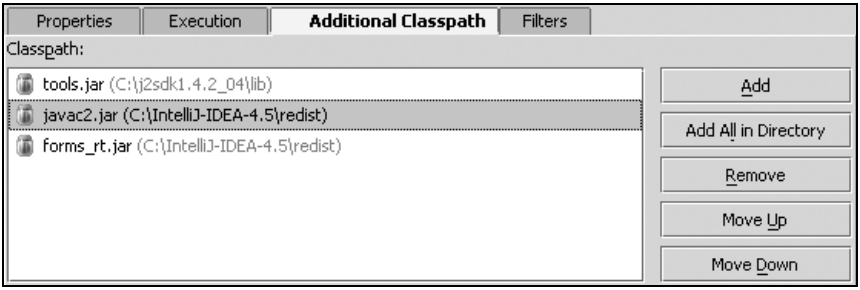


Рис. 13.17. Дополнительные библиотеки

Так, например, мы добавили в качестве дополнительной библиотеки файл javac2.jar из поставки IntelliJ IDEA, в котором содержится имплементация команды javac2 — расширенной версии стандартной команды javac, которая умеет компилировать файлы форм пользовательского интерфейса визуального редактора IntelliJ IDEA (*см. главу 7*).

С помощью кнопок в правой части диалогового окна можно:

- добавлять новые файлы библиотек (кнопка **Add**);
- добавлять все файлы библиотек из указанной директории (кнопка **Add All in Directory** (Добавить все из библиотеки));
- удалять файлы и директории библиотек (кнопка **Remove** (Удалить));
- изменять порядок библиотек (кнопки **Move Up/Down** (Переместить вверх/вниз)).

13.3.5. Фильтры

С помощью настройки фильтров, расположенных на последней закладке **Filters** (Фильтры), можно изменять количество отображаемых целей в списке целей Ant (рис. 13.18).

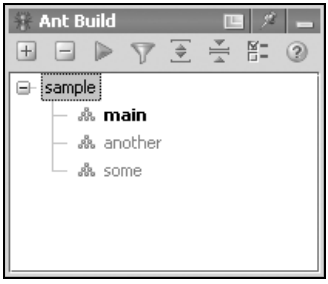


Рис. 13.18. Список целей из файла сборки Ant

На закладке содержится таблица, в которой представлены все цели из файла сборки Ant (рис. 13.19).

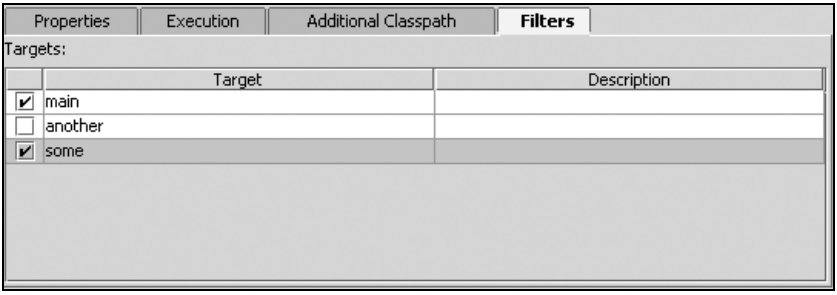


Рис. 13.19. Настройка фильтров

Перед каждой целью в таблице находится опция. Если включить эту опцию, то при фильтрации содержимого инструментального окна **Ant Build** (Сборка Ant), указанная цель останется в списке целей (рис. 13.20).

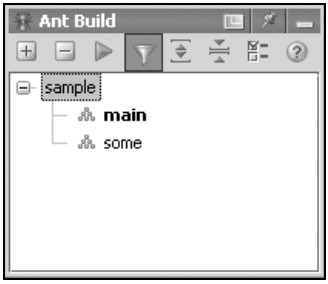


Рис. 13.20. Фильтрация целей

Фильтрация целей в списке производится с помощью кнопки **Filter targets** (Фильтровать цели) с панели инструментов окна **Ant Build** (Сборка Ant).

Возможность фильтрации весьма полезна, поскольку в файлах сборки может быть очень большое количество целей, ориентироваться среди которых не очень просто.

13.4. Привязка целей Ant

В IntelliJ IDEA имеется возможность соотнести выполнение отдельных целей из файла сборки Ant при выполнении следующих действий:

- сборка и компиляция проекта — цели Ant могут выполняться до или после компиляции;
- запуск/отладка приложения — цели Ant могут выполняться до запуска/отладки приложения.

Для настройки соответствия целей можно использовать команды контекстного меню **Execute on** (Выполнять для) инструментального окна **Ant Build** (Сборка Ant) (рис. 13.21):

- **Before Compilation** (Выполнять перед компиляцией);
- **After Compilation** (Выполнять после компиляции);
- **Before Run/Debug** (Выполнять перед запуском/отладкой).

Перед компиляцией и после компиляции может быть выполнена только одна цель — данные пункты меню работают как переключатели.

При выполнении цели перед запуском/отладкой для цели можно указать множество конфигураций, при запуске/отладке которых она будет выполняться (рис. 13.22).

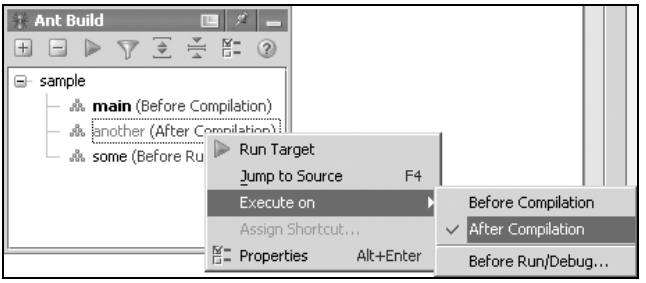


Рис. 13.21. Установка соответствия целей Ant

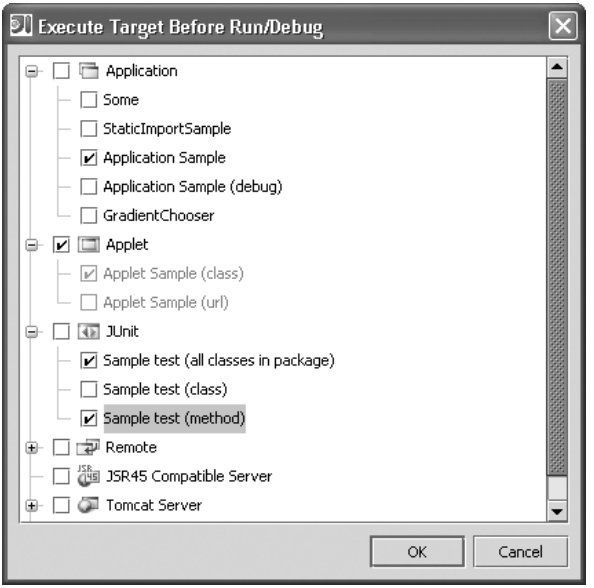


Рис. 13.22. Выбор конфигураций запуска и отладки, перед которыми должна выполняться цель Ant

13.5. Запуск целей Ant

Чтобы запустить некоторую цель Ant, необходимо выбрать ее в списке целей в инструментальном окне **Ant Build** (Сборка Ant) и выполнить для нее команду **Run** (Запуск) с панели инструментов, либо команду **Run Target** (Запустить цель) из контекстного меню инструментального окна **Ant Build** (Сборка Ant).

Результаты выполнения цели выводятся в инструментальном окне **Messages** (Сообщения) (рис. 13.23).

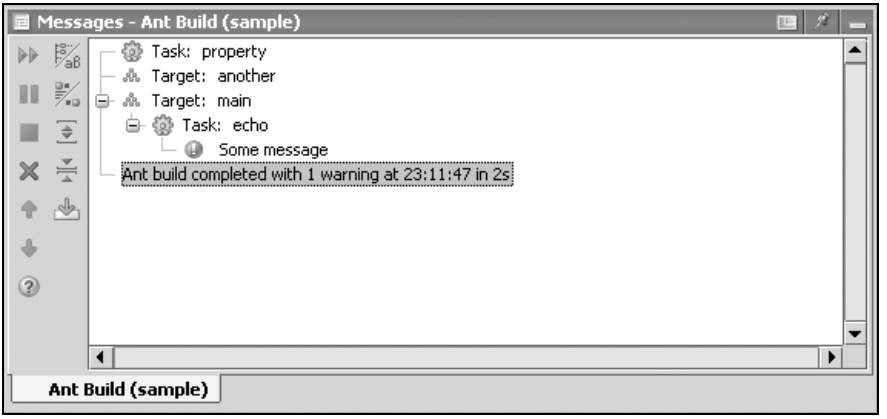


Рис. 13.23. Результат выполнения цели Ant

Результат выполнения может быть представлен в двух видах:

- в виде иерархического списка выполнения целей и команд (как на рис. 13.23);
- в виде обычного текста, как это обычно выглядит, если утилиту Ant запускают в консоли (рис. 13.24).

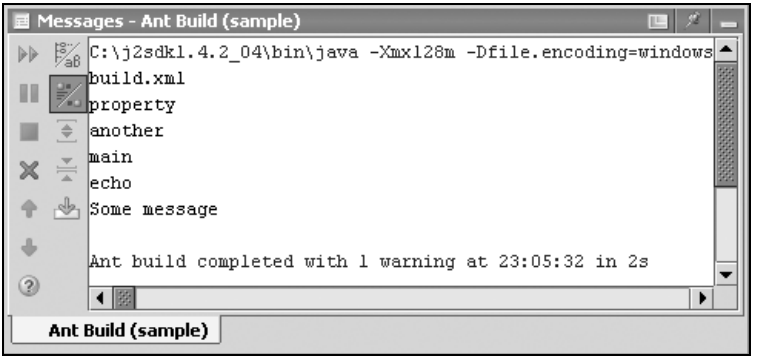


Рис. 13.24. Просмотр результатов работы Ant в виде обычного текста

Переключение между двумя этими режимами можно производить с помощью кнопки **Toggle tree/text mode** (Переключить режим дерево/текст) с панели инструментов окна **Messages** (Сообщения).

Также на панели инструментов имеется кнопка **Show All Messages** (Показать все сообщения), которая позволяет переключать просмотр результатов работы Ant между сокращенным (см. рис. 13.23) и полным видом (рис. 13.25).

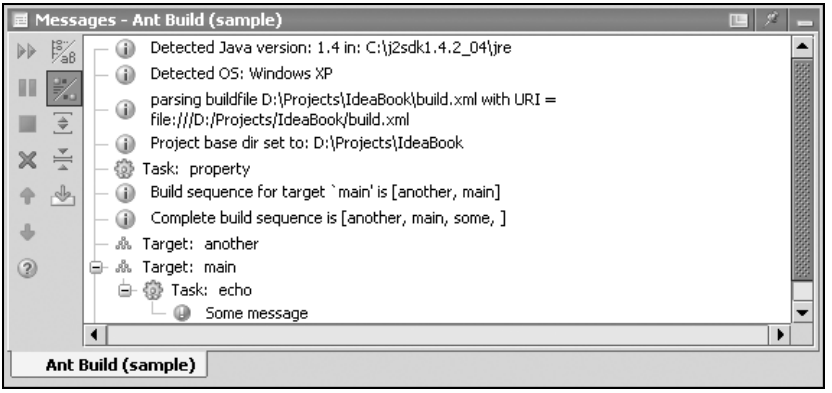


Рис. 13.25. Расширенный режим просмотра результатов вывода

13.6. Генерация файла сборки для проекта

IntelliJ IDEA позволяет создавать файл сборки Ant для имеющегося проекта. Этой возможностью удобно пользоваться для автоматизации сборки крупных проектов. Так, например, с помощью такого автоматически созданного файла собирается проект для самой IntelliJ IDEA.

Для создания файла сборки для проекта нужно использовать команду **Generate Ant Build** (Создать файл сборки Ant) из раздела **Build** (Сборка) главного меню IntelliJ IDEA (*см. раздел 3.2.9.10*).

Мы создали такой файл для нашего проекта (рис. 13.26).

Мы указали IntelliJ IDEA создать один файл для сборки всех модулей проекта. Можно заметить, что в полученном файле довольно много целей. Для каждого модуля создается четыре цели для компиляции основных файлов, компиляции тестов, общей компиляции и очистки рабочих файлов модуля.

В случае такого большого файла сборки удобно использовать фильтрацию целей (*см. раздел 13.3.5*) (рис. 13.27).

Данный файл сборки при запуске выполняет те же действия, что и команда сборки проектов из самой IntelliJ IDEA.

Если при сборке проекта с помощью файла не требуется компилировать файлы тестов, можно установить соответствующее свойство `skip.tests=true` либо при запуске сборки, либо прямо в файле сборки (рис. 13.28).

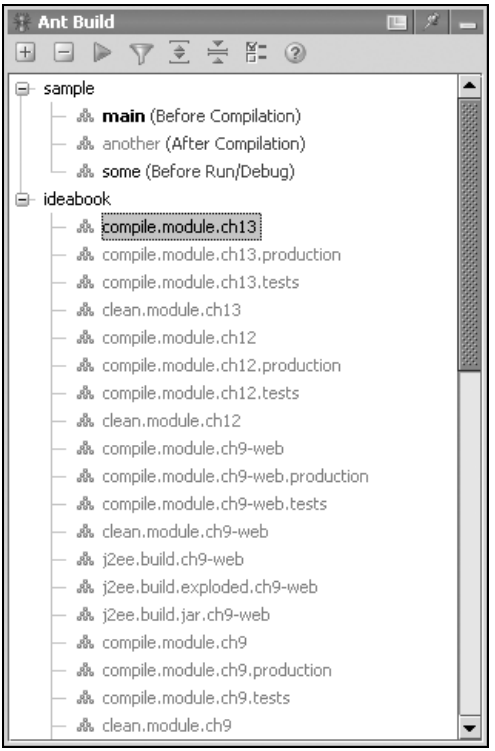


Рис. 13.26. Файл сборки проекта добавлен в общий список

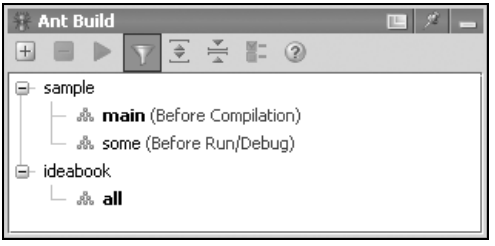


Рис. 13.27. Отфильтрованные цели

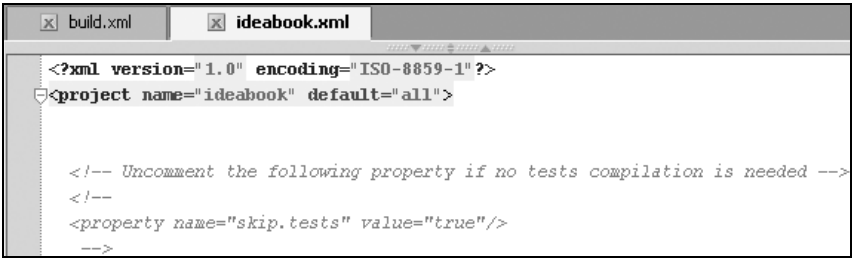
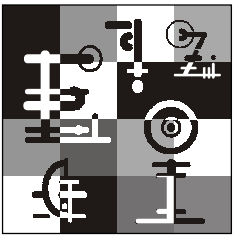


Рис. 13.28. Свойство исключения из сборки файлов с тестами



Глава 14

Тестирование

IntelliJ IDEA имеет встроенную поддержку unit-тестирования с использованием утилиты JUnit, которая поставляется совместно с библиотекой JUnit версии 3.8.1.

В данной главе будет рассказано об использовании тестов JUnit в рамках проекта, проведении тестирования, а также о просмотре результатов.

14.1. Представление тестов в проекте

Тесты JUnit представляют собой обычные Java-классы, наследуемые от класса `TestCase`. Данный класс должен содержать методы, начинающиеся со слова `test`, имеющие модификатор доступа `public`, не принимающие параметров и не возвращающие значений. В листинге 14.1 приведен пример простейшего теста (всегда успешно выполняющегося).

Листинг 14.1

```
import junit.framework.TestCase;


public class SimpleTest extends TestCase {
    public void test() {
        assertTrue("This test should be always successful", true);
    }
}
```

В рамках проекта тесты должны содержаться в специальных директориях с исходными текстами тестов (**Test Source Directory**) (рис. 14.1).

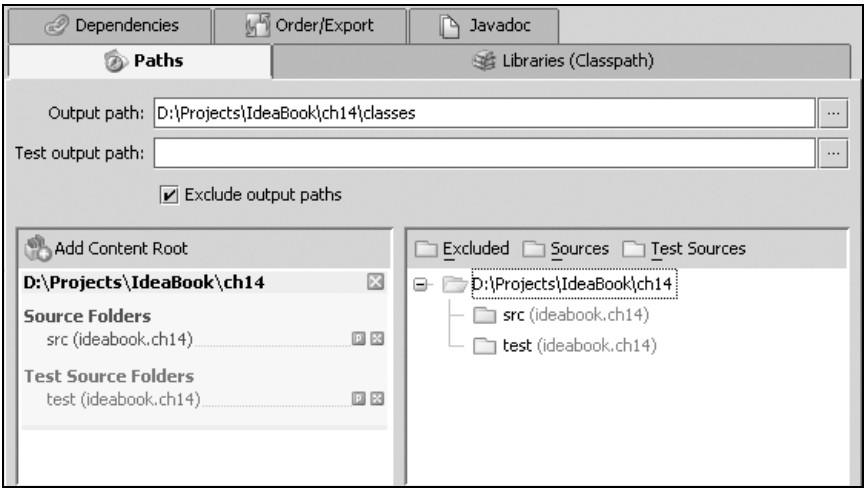


Рис. 14.1. Директория с исходными текстами тестов в настройках проекта

В принципе, тесты могут содержаться и в директориях с основными классами проекта, однако в этом случае вы не сможете воспользоваться некоторой дополнительной функциональностью, связанной со сборкой проекта (*см. раздел 13.6*). Может быть, это и не критично с точки зрения функциональности проекта, однако такое разделение может быть весьма полезным в крупных проектах с большим числом классов и тестов к ним.

Для того чтобы классы тестов успешно компилировались, в проект необходимо добавить библиотеку junit.jar, которую можно найти в директории lib/ стандартной установки IntelliJ IDEA.

14.2. Запуск тестов

Тесты JUnit можно запускать, как и обычные приложения на Java, с помощью метода `main`, например, как это сделано в классе из листинга 14.2.

Листинг 14.2

```
import junit.framework.TestSuite;
import junit.textui.TestRunner;

public class MyTestRunner {
    public static void main(String[] args) {
        new TestRunner().doRun(new TestSuite(SimpleTest.class));
    }
}
```

Однако в IntelliJ IDEA имеется более удобный способ запускать тесты JUnit. Для начала нужно создать конфигурацию для запуска тестов (*см. раздел 9.2.3*). Например, такую, как показано на рис. 14.2.

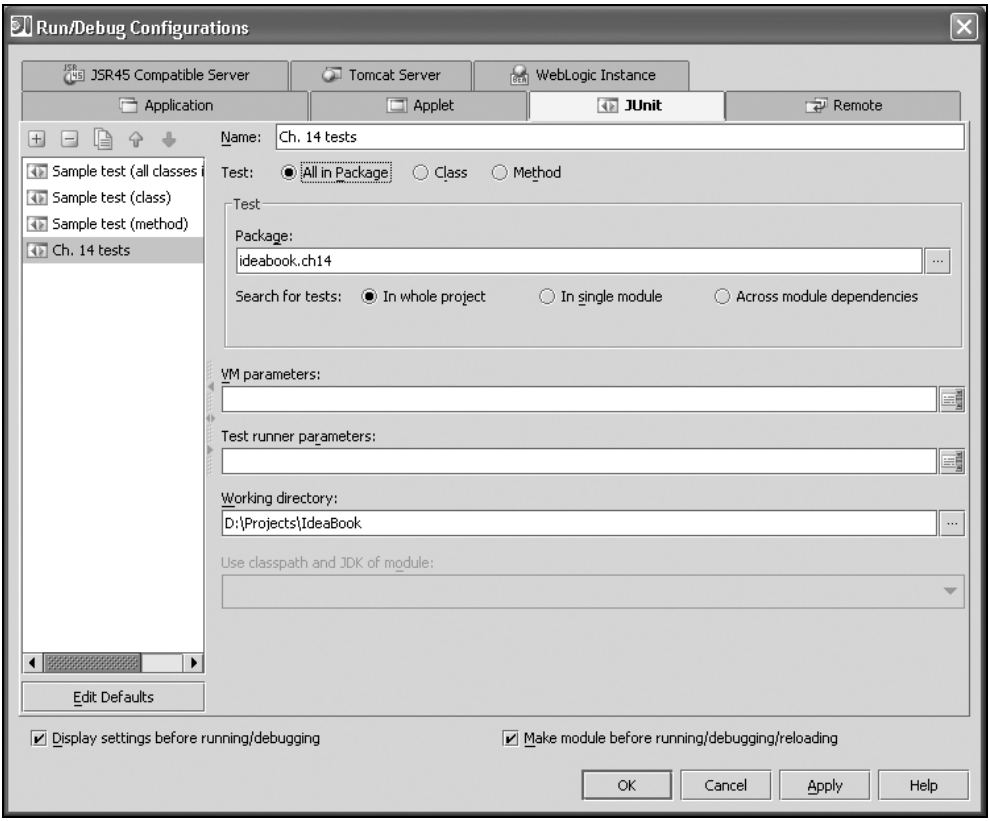


Рис. 14.2. Создание конфигурации запуска тестов

После чего, тесты можно будет запустить с помощью кнопки **Run** (Запуск) на панели инструментов IntelliJ IDEA (рис. 14.3).

Рис. 14.3. Фрагмент панели инструментов IntelliJ IDEA

Тесты также можно запустить в режиме отладки, что часто бывает полезным.

Рассмотрим пример тестирования класса из листинга 14.3.

Листинг 14.3

```
public class StringConcatenator {
    public static String concatenate (String... strings) {
        final StringBuffer buffer = new StringBuffer();
        for (String str : strings) {
            buffer.append(str);
        }
        return buffer.toString();
    }
}
```

В листинге 14.4 представлены тесты для данного класса (некоторые из них не вполне корректные, а один содержит явную ошибку).

Листинг 14.4

```
import junit.framework.TestCase;

public class ConcatenatorTest extends TestCase {
    private static final String STR_1 = "abc";
    private static final String STR_2 = "def";
    private static final String STR_3 = "ghi";
    private String concatenation;

    public void test1 () {
        concatenation = "abcdefghi";
        assertEquals("Concatenation of \"" + STR_1 +
                "\", \"" + STR_2 +
                "\" and \"" + STR_3 +
                "\" don't get the \"" + concatenation +
                "\"", concatenation,
                StringConcatenator.concatenate(
                   new String [] {STR_1, STR_2, STR_3}));
    }
    public void test2 () {
        concatenation = "ghidefabc";
        assertEquals("Concatenation of \"" + STR_3 +
                "\", \"" + STR_2 +
                "\" and \"" + STR_1 +
```

```
                "\" don't get the \"" + concatenation +
                "\"", concatenation,
                StringConcatenator.concatenate(
                   new String [] {STR_3, STR_2, STR_1}));
    }
    public void test3 () {
        concatenation = null;
        assertEquals("Concatenation of \"" + STR_2 +
                "\", \"" + STR_1 +
                "\" and \"" + STR_3 +
                "\" don't get the \"" + concatenation +
                "\"", concatenation,
                StringConcatenator.concatenate(
                   new String [] {STR_2, STR_1, STR_3}));
    }
    public void test4 () {
        concatenation = null;
        assertEquals("Concatenation of null don't get the null" +
                concatenation +
                "\"", concatenation,
                StringConcatenator.concatenate(null));
    }
}
```

Первые два теста должны проходить успешно, третий — не проходить, а четвертый заканчиваться ошибкой `NullPointerException`.

Запустим наши тесты.

Кстати, с помощью созданной конфигурации запуска также можно запускать тесты под отладчиком (*см. раздел 9.4*).

14.3. Просмотр результатов

После запуска тестов в инструментальном окне **Run** (Запуск) будут показаны результаты выполнения тестов (*см. раздел 9.3.3*).

В случае наших тестов результат представлен на рис. 14.4.

Как мы и говорили, два теста (`test1`, `test2`) прошли успешно, один из тестов (`test3`) не прошел, а последний (`test4`) — завершился с ошибкой.

В зависимости от результатов выполнения тестов они оказались помеченными значками, символизирующими их статус.

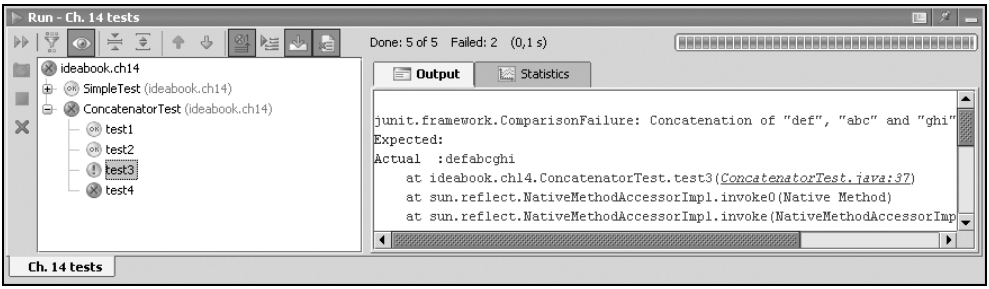


Рис. 14.4. Результат выполнения тестов

Если тест, который использует в качестве критерия прохождения один из методов `assertEquals()` (сравнение двух параметров), не прошел после запуска, то с помощью команды **View assertEquals Difference** (Посмотреть разницу значений `assertEquals`) из контекстного меню теста (см. рис. 14.4) можно сравнить ожидаемый и полученный результаты (рис. 14.5).

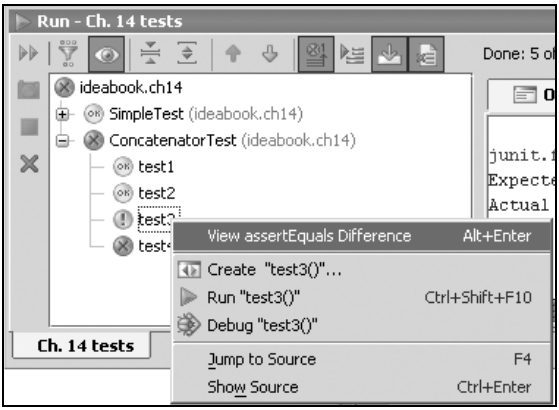


Рис. 14.5. Команда сравнения отличий для теста

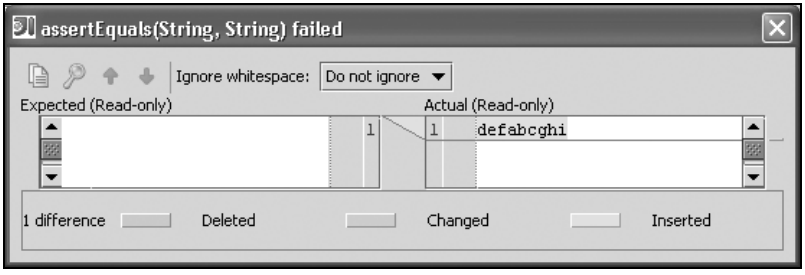
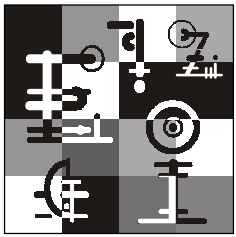


Рис. 14.6. Результат сравнения

На левой панели отображается ожидаемое значение, на правой — переданное в метод `assertEquals()` для сравнения.



Глава 15

Плагины

15.1. Общие сведения о плагинах

Плагин — подключаемый к основному приложению компонент или модуль.

В IntelliJ IDEA имеется ряд возможностей для расширения стандартной функциональности с помощью плагинов. Существует обширный репозиторий плагинов для IntelliJ IDEA как бесплатных, так и коммерческих. Репозиторий плагинов доступен из диалога настройки плагинов (*см. раздел 4.1.19*), а также по адресу **http://plugins.intellij.net**.

15.1.1. Сообщество разработчиков плагинов IDEA

В ходе развития интерфейса плагинов для IntelliJ IDEA возник вопрос о создании информационного ресурса, на котором разработчики плагинов смогли бы поделиться опытом, разместить свои плагины, создать и опубликовать документацию по применению, а также собирать отзывы и пожелания пользователей.

Был создан Wiki-ресурс — **http://www.intellij.org** (Wiki — технология, позволяющая накапливать знания в виде страниц, написанных специальным образом). Со временем там появилась простейшая система для поиска плагинов. Данный ресурс содержит очень много полезной информации как для разработчиков плагинов, так и для обычных пользователей IntelliJ IDEA. Кроме того, это единственное хранилище опыта почти пятилетней работы с IntelliJ IDEA по разработке плагинов и внутреннему устройству IntelliJ IDEA, а также по различным аспектам улучшения и расширения функций IntelliJ IDEA.

15.2. Разработка плагинов

Плагин в терминологии архитектуры IntelliJ IDEA — один или несколько компонентов различных уровней, объединенных при помощи XML-дескриптора, который будет рассмотрен в *разделе 15.4*. Разработка плагинов

в основном направлена на расширение функциональности приложения, однако сам плагин может являться прототипом отдельного продукта. И в том и в другом случае для его создания требуется открытый функциональный интерфейс, который в IntelliJ IDEA называется OpenAPI и представляет собой отдельную библиотеку. Библиотека OpenAPI представлена в виде архива openapi.jar в поддиректории lib корневой директории IntelliJ IDEA.

Исходный код библиотеки OpenAPI (Plugin Development Package) доступен как под платформой JDK 1.4, так и под платформой JDK 5.0. Платформа JDK 5.0 более предпочтительна для разработки плагинов, так как большая часть исходных кодов OpenAPI написана именно для JDK 5.0 и перенесена под JDK 1.4 лишь для совместимости. Прежде чем приступить к написанию своего первого плагина для IntelliJ IDEA, необходимо загрузить исходный код библиотеки OpenAPI (Plugin Development Package), который поставляется отдельно от основного дистрибутива IDEA. Интернет-адрес, по которому доступен архив с исходными кодами и JavaDoc-документацией библиотеки OpenAPI — **http://www.jetbrains.com/idea/download**. Данный архив также находится на прилагаемом диске в директории IDEA (файл idea4.5.3-dev.zip).

15.2.1. Подготовка IntelliJ IDEA к разработке плагинов

Подготовка IntelliJ IDEA к разработке плагинов состоит из следующих шагов:

- установка исходных кодов OpenAPI (*см. раздел 15.2.1.1*). Данный шаг рекомендуется выполнить;
- установка специального Ant-скрипта для автоматической сборки и отладки плагинов (*см. раздел 15.2.1.2*). Данный шаг является необязательным, так как разработку можно вести и без специального Ant-скрипта, но многие операции, такие как запуск отладочной копии IntelliJ IDEA и подготовка ZIP-архива, придется выполнять самостоятельно;
- настройка "песочницы" для отладки плагинов (*см. раздел 15.2.1.3*). Данный шаг можно пропустить, если предыдущий шаг выполнен, так как Ant-скрипт может создать "песочницу" автоматически;
- настройка глобальных библиотек (*см. раздел 15.2.1.4*).

Все эти шаги необходимы для настройки самой среды разработки, однако не являются достаточными, так как предстоит еще специальная настройка проекта. Подготовка проекта, в свою очередь, состоит из следующих шагов:

- настройка путей и библиотек;
- настройка компилятора;
- настройка отладчика.

15.2.1.1. Установка пакета Plugin Development Package

Для установки необходимо распаковать архив idea4.5.3-dev.zip в корневую директорию IntelliJ IDEA. Для разработки под JDK 1.4 потребуются исходные коды в архивах *без* суффикса `generics`, а для разработки под JDK 5.0, наоборот, именно *с* суффиксом `generics`.

JavaDoc-документация библиотеки OpenAPI помещена в архив **help/openapihelp.jar** относительно корневой директории IntelliJ IDEA.

Кроме исходных кодов самой библиотеки OpenAPI, в пакете находятся все исходные коды предустановленных плагинов, которые могут быть использованы в качестве примеров. Исходные коды плагинов находятся в поддиректории plugins корневой директории IntelliJ IDEA.

15.2.1.2. Установка Ant-скрипта для сборки и отладки плагинов

Для установки Ant-скрипта необходимо распаковать архив template-4.5.0.3.zip в рабочую директорию.

Интернет-адрес, по которому доступен данный архив — **http://www.intellij.org/twiki/bin/view/Main/%20t?topic=PluginAntScript**. Также данный архив может быть найден на прилагаемом к книге диске в директории /Projects/IDEABook/ch15/ant. Архив содержит шаблонный проектный Ant-скрипт и вспомогательный скрипт, с помощью которого осуществляется установка проектного скрипта в существующие проекты. Основные возможности вспомогательного скрипта:

- автоматическое обновление с сайта сообщества IntelliJ;
- автоматическая установка нового скрипта в существующий проект плагина.

Основные возможности проектного скрипта:

- сборка плагина в промежуточной версии, для последующей отладки;
- запуск отладочной версии плагина в "песочнице";
- сборка дистрибутивной версии плагина для последующей загрузки в репозиторий плагинов.

Работа со вспомогательным скриптом

С помощью вспомогательного скрипта можно обновлять текущий дистрибутив, а также устанавливать проектные скрипты в существующие проекты.

При первом запуске скрипта будет запущен настроечный сценарий, который соберет основную информацию, необходимую для корректной работы.

Примечание

Настроечный сценарий может быть вызван принудительно, с помощью команды `ant configure` в директории, куда был распакован архив с Ant-скриптом.

Сценарий соберет следующие сведения:

- информацию о том, где на диске находится IntelliJ IDEA;
- персональную информацию об авторе плагинов, для автоматической подстановки в XML-дескриптор плагина;
- адреса автоматического обновления из Интернета. Данные адреса вряд ли придется менять, однако такая возможность существует.

Для установки нового проектного скрипта в существующий проект выполните команду `ant install` в директории, куда был распакован архив с Ant-скриптом. Скрипт запросит полный путь до проектной директории, после чего начнет копирование файлов. Обычно после данной операции необходимо произвести несколько изменений в проекте и, возможно, в Ant-скрипте:

- необходимо подключить файл build.xml как Ant-скрипт к проекту;
- если в проекте используются визуальные формы, созданные в редакторе визуальных форм IntelliJ IDEA, то необходимо добавить в пути запуска Ant-скрипта библиотеку javac2.jar из поддиректории redist и библиотеки jdom.jar и bcel.jar из поддиректории lib корневой директории IntelliJ IDEA;
- файл build.properties содержит ряд полезных настроек для скрипта и должен быть скорректирован сразу после установки.

Работа с проектным скриптом

Перед началом использования скрипта необходимо произвести настройки, если это необходимо. В начале скрипта приведены настройки путей проекта, кроме того, в файл build.properties вынесены персональные данные автора, информация о версии плагина, название и описание плагина, шаблоны наименования выходных ZIP-архивов и т. п.

Для работы со скриптом необходимо использовать цель `deploy` при отладке очередных изменений в проекте. Когда все изменения примут стабильный версионный характер, возможно, понадобится выложить эту версию в репозиторий плагинов, тогда должна быть использована цель `release`. Подробнее о работе с файлами сборки Ant рассказано в *главе 13*.

Для запуска цели `release` необходимо заполнить файл changes.log. В данном файле указываются изменения, которые были сделаны с момента последней публикации плагина. Каждая строчка файла changes.log соответствует одной записи об изменении. Если было сделано несколько изменений, то и строчек должно быть несколько.

Скрипт может использовать подстановочные строки, которые обрамляются символами @. Так, например, если в XML-дескрипторе плагина вместо версии указана строка @plugin.version@, то при сборке Ant-скрипт заменит эту строку на значение версии, указанной в файле build.properties.

История изменений также генерируется автоматически на основе шаблонов, которые расположены в поддиректории ant корневой директории проекта, и помещается в дистрибутивный ZIP-архив.

15.2.1.3. Настройка "песочницы" для отладки плагинов

Для отладки плагинов необходимо подготовить "песочницу" (sandbox), где они могут "копошиться", не поломав рабочую инсталляцию IntelliJ IDEA.

Внимание

Необходимость такой "песочницы" осознается, к сожалению, лишь в тот момент, когда отлаживаемый плагин даст сбой при загрузке проекта, и IntelliJ IDEA не сможет сохранить настройки проекта. В результате придется настраивать заново весь проект. Точно так же могут пропасть и все настройки из конфигурационных файлов IntelliJ IDEA. Поэтому рекомендуется не пренебрегать настройкой отдельной "песочницы" и не пытаться отлаживать плагин на той же конфигурации, на которой происходит основная разработка.

"Песочница" — это директория, как правило, внутри корневой директории IntelliJ IDEA. В нее необходимо копировать плагин для отладки, после чего запускать еще одну копию IntelliJ IDEA, настроенную специальным образом на данную директорию. Для создания "песочницы" необходимо выполнить следующие шаги:

- создать директорию sandbox внутри корневой директории IntelliJ IDEA;
- в поддиректории bin корневой директории IntelliJ IDEA создать файл запуска для "песочницы" — sandbox.bat (листинг 15.1).

Листинг 15.1

```
@echo off

setlocal
set IDEA_HOME=..
set JAVA_EXE=%IDEA_HOME%\jre\bin\java.exe
if not exist "%JAVA_EXE%" goto error

set JVM_ARGS=-Xms32m
set JVM_ARGS=%JVM_ARGS% -Xmx192m
```

```
set JVM_ARGS=%JVM_ARGS% -Dsun.java2d.noddraw=true
set JVM_ARGS=%JVM_ARGS% -Xdebug
set JVM_ARGS=%JVM_ARGS% -Xnoagent
set JVM_ARGS=%JVM_ARGS% -Djava.compiler=NONE
set JVM_ARGS=%JVM_ARGS% -Xrunjdwp:transport=dt_shmem
set JVM_ARGS=%JVM_ARGS%,server=y
set JVM_ARGS=%JVM_ARGS%,suspend=n
set JVM_ARGS=%JVM_ARGS%,address=ideapluginsdebug
set JVM_ARGS=%JVM_ARGS% -Didea.system.path=%IDEA_HOME%\sandbox\system
set JVM_ARGS=%JVM_ARGS% -Didea.config.path=%IDEA_HOME%\sandbox\config
set JVM_ARGS=%JVM_ARGS% -Didea.plugins.path=%IDEA_HOME%\sandbox\plugins
set JVM_ARGS=%JVM_ARGS% -Didea.popup.weight=heavy

set PATH=%IDEA_HOME%\bin;%PATH%

set CLASS_PATH=%IDEA_HOME%\lib\idea.jar
set CLASS_PATH=%CLASS_PATH%;%IDEA_HOME%\lib\openapi.jar
set CLASS_PATH=%CLASS_PATH%;%IDEA_HOME%\lib\jdom.jar
set CLASS_PATH=%CLASS_PATH%;%IDEA_HOME%\lib\log4j.jar

%JAVA_EXE% %JVM_ARGS% -cp "%CLASS_PATH%" com.intellij.idea.Main %*

goto end
:error
echo ------------------------------------------------------------------------
echo ERROR: cannot start IntelliJ IDEA.
echo No JRE found in IDEA installation directory
echo ------------------------------------------------------------------------
pause
:end
endlocal
```

Внимание

Запуск осуществляется в режиме отладки, и `ideapluginsdebug` — это адрес области памяти для удаленной отладки плагинов. Свойства `idea.system.path`, `idea.config.path` и `idea.plugins.path` указывают на соответствующие директории, которые должны находиться внутри директории sandbox. Запуск файла sandbox.bat должен осуществляться из директории bin, находящейся в корневой директории IntelliJ IDEA.

15.2.1.4. Настройка глобальных библиотек

Для разработки плагинов могут потребоваться библиотеки, часть которых уже есть в составе IntelliJ IDEA:

- библиотека Log4J. Данная библиотека понадобится для диагностики;
- библиотека JDOM. Данная библиотека необходима для работы с XML, а также для работы с конфигурационным функционалом IntelliJ IDEA;
- библиотека JUnit. Данная библиотека понадобится для unit-тестирования плагинов, если вы захотите написать unit-тесты для ваших классов;
- библиотека BCEL. Данная библиотека содержит функции для работы с byte-кодом. Может понадобиться при выполнении инструментального компилирования классов, а также для анализа API-скомпилированных классов;
- библиотека Trove4J. Данная библиотека должна быть включена в проект, если вы хотите использовать UI-компоненты из библиотеки OpenAPI. Библиотека содержит коллекции, расширяющие коллекции JDK.

Библиотеки OpenAPI и Forms Runtime являются частью IntelliJ IDEA. Пакет для разработки плагинов содержит исходные коды для данных библиотек.

- Библиотека OpenAPI содержит основные классы и интерфейсы для разработки плагинов.
- Библиотека Forms Runtime необходима для запуска форм, созданных в визуальном редакторе форм IntelliJ IDEA.

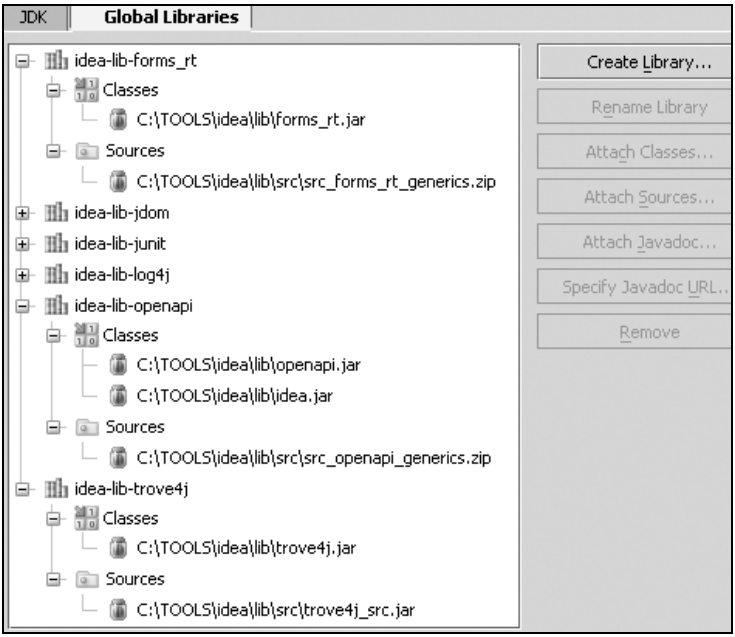


Рис. 15.1. Глобальные библиотеки для разработки плагинов

Перейдите к настройке глобальных библиотек IntelliJ IDEA (<Ctrl>+<Alt>+<S>, <H>). Щелкните по вкладке **Global Libraries** (Глобальные библиотеки) (*см. раздел 4.1.8.2*) (рис. 15.1).

Совет

При создании библиотек рекомендуется использовать префикс `idea-lib`, это поможет отличить, например, библиотеку Log4J из дистрибутива IntelliJ IDEA от другой версии библиотеки Log4J, которая, возможно, имеется на вашем диске.

Табл. 15.1 приведена для сопоставления названий библиотек с JAR-файлами и архивами с исходными кодами.

Таблица 15.1. *Глобальные библиотеки*

Библиотека	Имя JAR-файла	Имя архива с исходными кодами
idea–lib–forms_rt	forms_rt.jar	src/src_forms_rt_generics.zip для JDK 5.0
		src/src_forms_rt.zip для JDK 1.4
idea–lib–jdom	jdom.jar	—
idea–lib–junit	junit.jar	—
idea–lib–log4j	log4j.jar	—
idea–lib–openapi	openapi.jar	src/src_opeanapi_generics.zip для JDK 5.0
	idea.jar	src/src_opeanapi.zip для JDK 1.4
idea–lib–trove4j	trove4j.jar	src/trove4j_src.zip

Примечание

Файл idea.jar содержит закрытый функционал и его использование может привести к несовместимости версий, однако он может быть полезен в некоторых случаях, так как содержит набор иногда просто незаменимых классов, в том числе и UI-компонент для визуального редактора IntelliJ IDEA.

15.2.1.5. Настройка проекта

Настройка путей и библиотек

Настройки для Java-модуля совершенно стандартны, как и для любого обычного проекта (рис. 15.2). Единственное, на что следует обратить внимание, — это то, что сборка проекта осуществляется Ant-скриптом, поэтому путь для сохранения скомпилированных классов указывает на ту директорию, которая используется при сборке Ant-скриптом. Сама же директория build исключена из проекта, чтобы не заставлять IntelliJ IDEA индексировать изменения в ней при каждой сборке. Директория lib исключена из

проекта всего лишь для удобства. При работе с проектом, в котором задействован версионный контроль, не рекомендуется исключать директорию lib.

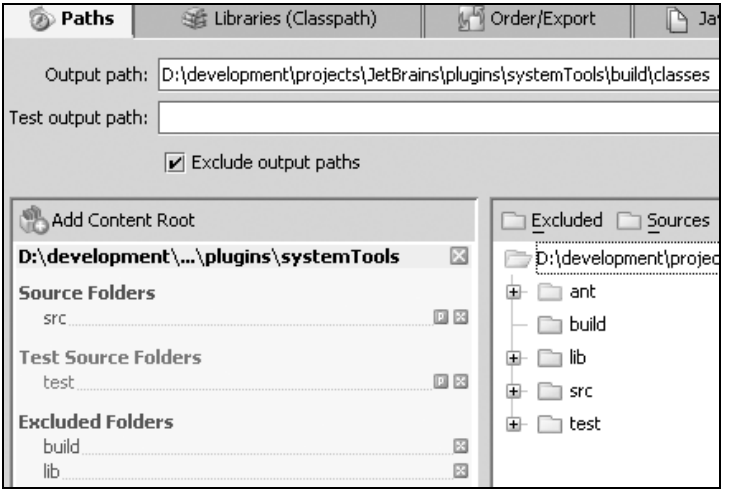


Рис. 15.2. Вид окна настроек Java-модуля для плагина

Директория lib содержит проектные библиотеки. Настройка библиотек проекта заключается в добавлении библиотек в виде проектных или модульных библиотек, а также в подключении глобальных библиотек (рис. 15.3).

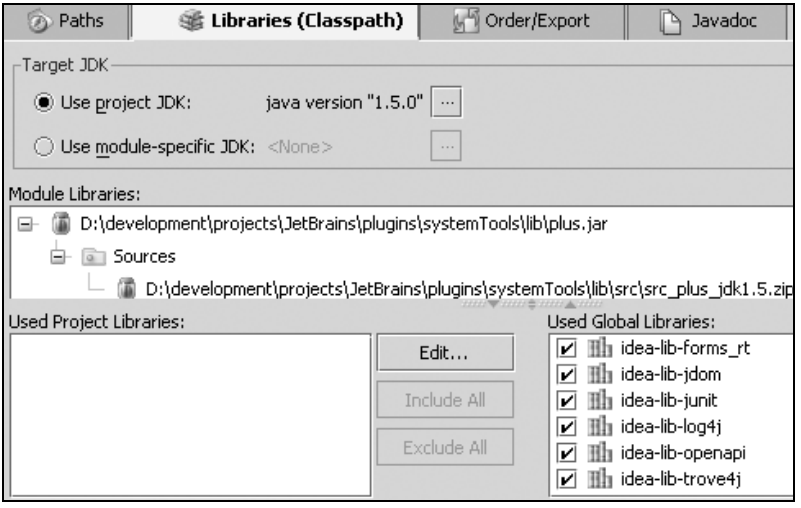


Рис. 15.3. Вид окна настроек библиотек Java-модуля

Настройка компилятора для JDK 5.0

Если вы ведете разработку плагина под JDK 5.0, вам необходимо соблюдать совместимость версий JDK, так как IntelliJ IDEA работает под управлением JRE 1.4. Поэтому дистрибутив плагина должен содержать совместимый с JDK 1.4 byte-код. Для этого задайте параметр компилятора `-target jsr14` в настройках компилятора проекта (*см. раздел 4.2.2*) (рис. 15.4).

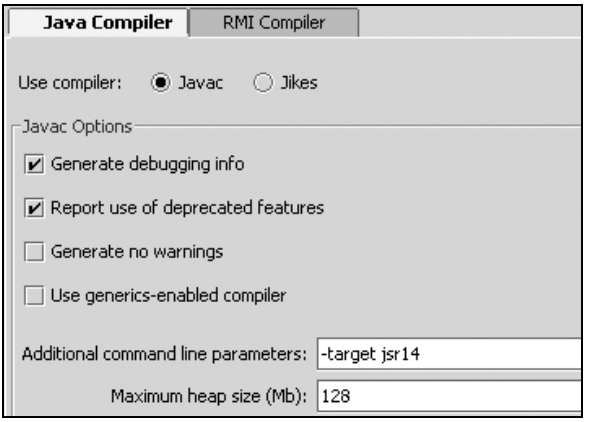


Рис. 15.4. Вид окна настроек компилятора для JDK 5.0

Настройка отладчика

Для настройки отладчика необходимо вызвать конфигурацию отладчика. Для этого перейдите в меню **Run** (Запуск) и выберите команду **Edit Configurations** (Редактировать конфигурации) (*см. раздел 3.2.10.1*) (рис. 15.5). Перейдите на

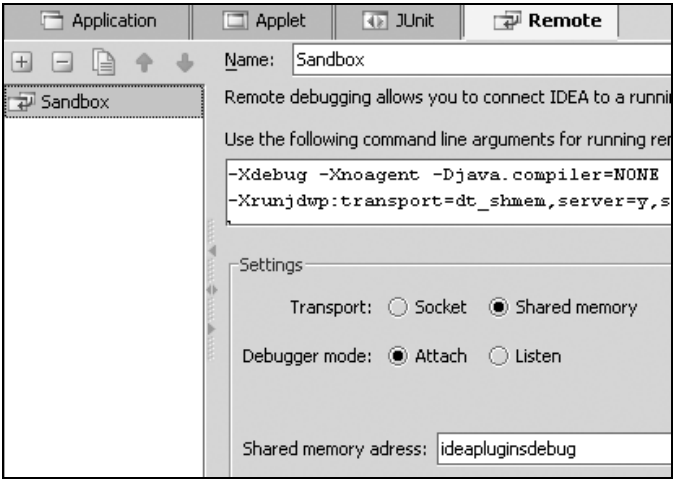


Рис. 15.5. Настройка удаленного отладчика

закладку **Remote** (Конфигурация для удаленной отладки) и добавьте новую конфигурацию для удаленной отладки. Назовите новую конфигурацию — Sandbox. Для транспортного протокола (**Transport**) используйте **Shared memory** (Разделяемая память), а в качестве адреса впишите строку `ideapluginsdebug`, это та строка, которую вы указали в файле sandbox.bat (см. листинг 15.1). Режим отладки (**Debugger mode**) должен быть установлен в значение **Attach** (Присоединение к запущенной виртуальной машине) (*см. раздел 9.2.4*).

15.2.1.6. Подготовка Ant-скрипта

Хотя для IntelliJ IDEA наличие Ant-скрипта для сборки плагина необязательно, я все же рекомендую это сделать. Это облегчит дальнейшую сборку и упаковку плагина в ZIP-архив для публикации. Список некоторых требований для Ant-скрипта типичного проекта плагина:

- компиляция должна соответствовать настройкам компилятора вашего проекта. Если вы разрабатываете под JDK 5.0, не стоит забывать о том, что и сама утилита Ant должна собирать проект с учетом совместимости с JDK 1.4 (*см. раздел 15.2.1.5*);
- если вы используете визуальный редактор IntelliJ IDEA (GUI designer), ваш Ant-скрипт должен компилировать и ваши визуальные формы. Это можно сделать с помощью специального расширения для Ant, которое входит в состав IntelliJ IDEA (*см. раздел 7.4.3*);
- если вы используете визуальный редактор IntelliJ IDEA, ваш Ant-скрипт должен копировать библиотеку forms_rt.jar в директорию lib дистрибутива плагина, иначе возникнут проблемы с формами, в которых применяется локализация;
- если вы используете ресурсы локализации (файлы, в которых присутствуют символы, выходящие за диапазон US ASCII), вам необходимо применять преобразование native2ascii для этих файлов (*см. раздел 7.4.4*);
- если, помимо библиотек, находящихся в поддиректории lib корневой директории IntelliJ IDEA, для плагина необходимы еще сторонние библиотеки, то все они должны быть скопированы при сборке в директорию lib дистрибутива плагина;
- при сборке было бы неплохо, если бы Ant-скрипт сам упаковывал дистрибутив в ZIP-архив, который может быть опубликован в репозитории плагинов.

Совет

Вы можете использовать готовый шаблонный скрипт в качестве Ant-скрипта для сборки плагина (*см. раздел 15.2.1.2*).

15.2.2. Общая компонентная модель

Архитектура IntelliJ IDEA имеет многоуровневую компонентную структуру. На самом нижнем уровне работают компоненты приложения. На следующем уровне обитают проектные компоненты, которые ассоциируются со специфичными для проекта атрибутами. А на самом верхнем уровне находятся модульные компоненты, которые ассоциируются уже с атрибутами модулей проекта.

Поэтому получение компонент, задействованных или в приложении, или в проекте, или в модуле зависит, в первую очередь, от того, какой является эта компонента и на каком уровне она работает. Так, проектная компонента может быть получена только в рамках проекта, а модульная — только в рамках модуля.

Компоненты уникальны на своих уровнях. Это означает, что для одного и того же проекта существует одна-единственная проектная компонента заданного типа, однако в рамках всего приложения проектных компонент одного типа может быть несколько. Для одного и того же модуля существует лишь одна-единственная модульная компонента заданного типа, однако в рамках проекта, а также в рамках всего приложения таких компонент может быть несколько.

Для получения компонент используется интерфейс `ComponentManager` из пакета `com.intellij.openapi.components`. Данный интерфейс наследуют все три уровня архитектурной модели:

- интерфейс `Application` (пакет `com.intellij.openapi.application`);
- интерфейс `Project` (пакет `com.intellij.openapi.project`);
- интерфейс `Module` (пакет `com.intellij.openapi.module`).

Для получения этих интерфейсов обычно используется менеджер из того же пакета (листинг 15.2).

Листинг 15.2. Получение компоненты из уровня приложения

```
Application application = ApplicationManager.getApplication();
Options options = (Options)application.getComponent(Options.class);
```

Совет

Существует ряд служебных компонент в IntelliJ IDEA, которые предоставляют различные сервисы для разработчиков плагинов. Среди них есть, например, компоненты `DimensionService` и `PropertiesComponent`, с помощью которых осуществляется сохранение размеров и положения диалоговых окон. Вы можете узнать все открытые компоненты, поискав имплементации интерфейса `BaseComponent` из пакета `com.intellij.openapi.components` в библиотеках проекта (*см. раздел 3.2.5.6*).

Любая компонента является имплементацией интерфейса `BaseComponent` из пакета `com.intellij.openapi.components`, где определены три метода:

- метод `getComponentName`. Этот метод должен возвращать служебный строковый идентификатор компоненты, который может быть использован при записи компоненты в XML-файл или при чтении компоненты из XML-файла. Данный идентификатор должен быть уникальным;
- метод `initComponent`. Данный метод вызывается, чтобы оповестить компоненту о том, что система производит его инициализацию. Инициализация компонент, находящихся на разных уровнях, производится в разные моменты времени, например, для уровня приложения все компоненты инициализируются при загрузке среды IntelliJ IDEA. Компоненты проектного уровня инициализируются при загрузке проекта, а компоненты модульного уровня при инициализации модулей проекта;
- метод `disposeComponent`. Данный метод вызывается, чтобы сигнализировать компоненте о том, что система освобождает компоненты данного уровня, и следует освободить использовавшиеся ресурсы, если это необходимо. Освобождение компонент приложения происходит при закрытии приложения IntelliJ IDEA. Компоненты проектного уровня освобождаются при закрытии проекта, как и компоненты модульного уровня, так как модули закрываются вместе с проектом.

15.2.2.1. Компоненты приложения

Компонентой приложения называется имплементация интерфейса `ApplicationComponent` из пакета `com.intellij.openapi.components`. Компонента приложения инициализируется в момент запуска IntelliJ IDEA и освобождается, когда приложение будет закрыто.

Компоненты приложения доступны сразу после запуска IntelliJ IDEA. Однако не следует пытаться в момент создания экземпляра компоненты получить другую компоненту среды, так как она скорей всего еще не будет загружена.

Процесс загрузки компонент приложения следующий:

1. Построение списка компонент. Находятся все компоненты, которые могут быть загружены.
2. Компоненты последовательно создаются с помощью вызова конструкторов *без параметров*.
3. Если компонента реализует интерфейс `JDOMExternalizable`, вызывается метод `readExternal`, с помощью которого происходит восстановление состояния компоненты из XML-файла.
4. Вызывается метод `initComponent` последовательно для всех компонент. Если в данном методе компонента пытается получить доступ к другой компоненте, которая еще не была проинициализирована, то IntelliJ IDEA начнет инициализацию данной компоненты по запросу.

Процесс выгрузки компонент приложения:

1. Если компонента реализует интерфейс `JDOMExternalizable`, то вызывается метод `writeExternal`, с помощью которого происходит сохранение состояния компоненты в XML-файл.
2. Вызывается метод `disposeComponent` последовательно для всех загруженных и проинициализированных компонент.

Пример компоненты уровня приложения представлен в листинге 15.3.

Листинг 15.3

```
package org.intellij.samples.components;

import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.diagnostic.Logger;

public class SampleApplicationComponent
    implements ApplicationComponent {
    private static final String COMPONENT_NAME =
        "SampleApplicationComponent";
    private static final Logger LOGGER = Logger.getInstance(
        COMPONENT_NAME
    );

    public String getComponentName() {
        return COMPONENT_NAME;
    }

    public void initComponent() {
        LOGGER.info("Init component");
    }

    public void disposeComponent() {
        LOGGER.info("Dispose component");
    }
}
```

15.2.2.2. Проектные компоненты

Проектной компонентой называется имплементация интерфейса `ProjectComponent` из пакета `com.intellij.openapi.components`. Компонен-

та инициализируется в момент загрузки проекта и освобождается после закрытия проекта. С проектной компонентой связываются специфичные для проекта атрибуты.

Процесс загрузки проектных компонент следующий:

1. Построение списка компонент. Находятся все компоненты, которые могут быть загружены.
2. Создание экземпляра компоненты конструктором без параметров или конструктором с одним параметром типа `Project`. Предпочтение отдается последнему типу конструктора, если представлены оба.
3. Если компонента реализует интерфейс `JDOMExternalizable`, вызывается метод `readExternal`, с помощью которого происходит восстановление состояния компоненты из XML-файла.
4. Вызывается метод `initComponent` последовательно для всех компонент. Если в данном методе компонента пытается получить доступ к другой компоненте, которая еще не была проинициализирована, то IntelliJ IDEA начнет инициализацию данной компоненты по запросу.
5. Вызывается метод `projectOpened` последовательно для всех компонент, чтобы сигнализировать об окончании загрузки проекта.

Процесс выгрузки проектных компонент:

1. Вызывается метод `projectClosed` последовательно для всех компонент, чтобы сигнализировать о закрытии проекта.
2. Если компонента реализует интерфейс `JDOMExternalizable`, вызывается метод `writeExternal`, с помощью которого происходит сохранение состояния компоненты в XML-файл.
3. Вызывается метод `disposeComponent` последовательно для всех загруженных и проинициализированных компонент.

Пример компоненты уровня проекта представлен в листинге 15.4.

Листинг 15.4

```
package org.intellij.samples.components;

import com.intellij.openapi.components.ProjectComponent;
import com.intellij.openapi.diagnostic.Logger;
import com.intellij.openapi.project.Project;

public class SampleProjectComponent implements ProjectComponent {
    private static final String COMPONENT_NAME =
```

```
        "SampleProjectComponent";
    private static final Logger LOGGER = Logger.getInstance(
        COMPONENT_NAME
    );

    public SampleProjectComponent(Project project) {
        LOGGER.info(
            "Project component created with " +
            project.getProjectFile()
        );
    }

    public void projectOpened() {
        LOGGER.info("Project opened");
    }

    public void projectClosed() {
        LOGGER.info("Project closed");
    }

    public String getComponentName() {
        return COMPONENT_NAME;
    }

    public void initComponent() {
        LOGGER.info("Init component");
    }

    public void disposeComponent() {
        LOGGER.info("Dispose component");
    }
}
```

15.2.2.3. Модульные компоненты

Модульной компонентой называется имплементация интерфейса `ModuleComponent` из пакета `com.intellij.openapi.module`. Компонента инициализируется в момент загрузки проекта и освобождается после закрытия проекта. С модульной компонентой связываются специфичные для модуля атрибуты.

Процесс загрузки модульных компонент следующий:

1. Построение списка компонент. Находятся все компоненты, которые могут быть загружены.
2. Создание экземпляра компоненты конструктором без параметров или конструктором с одним параметром типа `Module`. Предпочтение отдается последнему типу конструктора, если представлены оба.
3. Если компонента реализует интерфейс `JDOMExternalizable`, вызывается метод `readExternal`, с помощью которого происходит восстановление состояния компоненты из XML-файла.
4. Вызывается метод `initComponent` последовательно для всех компонент. Если в данном методе компонента пытается получить доступ к другой компоненте, которая еще не была проинициализирована, то IntelliJ IDEA начнет инициализацию данной компоненты по запросу.
5. Вызывается метод `moduleAdded` последовательно для всех компонент, чтобы сигнализировать о добавлении модуля в проект для загрузки.
6. Вызывается метод `projectOpened` последовательно для всех компонент, чтобы сигнализировать об окончании загрузки проекта.

Процесс выгрузки модульных компонент:

1. Вызывается метод `projectClosed` последовательно у всех компонент, чтобы сигнализировать о закрытии проекта.
2. Если компонента реализует интерфейс `JDOMExternalizable`, то вызывается метод `writeExternal`, с помощью которого происходит сохранение состояния компоненты в XML-файл.
3. Вызывается метод `disposeComponent` последовательно для всех загруженных и проинициализированных компонент.

Пример модульной компоненты представлен в листинге 15.5.

Листинг 15.5

```
package org.intellij.samples.components;

import com.intellij.openapi.diagnostic.Logger;
import com.intellij.openapi.module.Module;
import com.intellij.openapi.module.ModuleComponent;

public class SampleModuleComponent implements ModuleComponent {
    private static final String COMPONENT_NAME = "SampleModuleComponent";
    private static final Logger LOGGER = Logger.getInstance(
        COMPONENT_NAME
```

```
    );

    public SampleModuleComponent(Module module) {
        LOGGER.info(
            "Module component created with " +
            module.getModuleFile()
        );
    }

    public void projectOpened() {
        LOGGER.info("Project opened");
    }

    public void projectClosed() {
        LOGGER.info("Project closed");
    }

    public void moduleAdded() {
        LOGGER.info("Module added");
    }

    public String getComponentName() {
        return COMPONENT_NAME;
    }

    public void initComponent() {
        LOGGER.info("Init component");
    }

    public void disposeComponent() {
        LOGGER.info("Dispose component");
    }
}
```

15.2.3. Диагностика ошибок

Очень часто при разработке плагинов встает вопрос о диагностировании различных состояний в момент выполнения. Для этих целей в библиотеке OpenAPI предусмотрен абстрактный класс `Logger` из пакета `com.intellij.openapi.diagnostic`, являющийся оберткой библиотеки Log4J.

Для инициализации объекта `Logger` необходимо указать категорию, в которой будут транслироваться диагностические сообщения. Это может быть любая строка, однако целесообразно использовать в имени категории имя текущего класса, чтобы сообщения имели различимый префикс. В примере из листинга 15.6 имя категории задается на основе имени класса.

Листинг 15.6

```
package org.intellij.samples.diagnostics;

import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.diagnostic.Logger;

public class SampleLoggerComponent implements ApplicationComponent {
    private static final String COMPONENT_NAME = "SampleLoggerComponent";
    private static final Logger LOGGER = Logger.getInstance(
        COMPONENT_NAME
    );

    public String getComponentName() {
        return COMPONENT_NAME;
    }

    public void initComponent() {
        LOGGER.info("initComponent");
        try {
            throw new RuntimeException("Assertion");
        } catch (Exception e) {
            LOGGER.error(e);
        }
    }

    public void disposeComponent() {
        LOGGER.info("disposeComponent");
    }
}
```

Данный пример показывает варианты диагностирования ошибок.

Метод `error` с параметром типа `Throwable` на EAP-версии (*см. раздел 2.1*) IntelliJ IDEA выведет сообщение об ошибке в диалоговом окне (рис. 15.6). На стабильных версиях IntelliJ IDEA окно с ошибкой подавляется, и понять, что случилась ошибка, можно только по файлу idea.log, который находится в системной директории, например, sandbox/system/log (листинг 15.7).

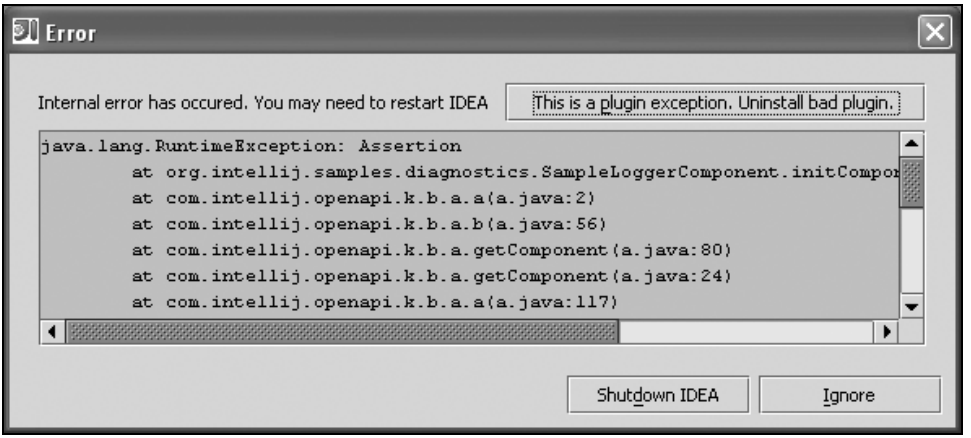


Рис. 15.6. Сообщение об ошибке в EAP-версии IntelliJ IDEA

Листинг 15.7

```
INFO — SampleLoggerComponent - initComponent
ERROR — SampleLoggerComponent -
ERROR — SampleLoggerComponent - IntelliJ IDEA 4.5.1 Build #2242
ERROR — SampleLoggerComponent - JDK: 1.4.2_04
ERROR — SampleLoggerComponent - VM: Java HotSpot(TM) Client VM
ERROR — SampleLoggerComponent - Vendor: Sun Microsystems Inc.
ERROR — SampleLoggerComponent - OS: Windows XP
ERROR — SampleLoggerComponent - Last Action:
ERROR — SampleLoggerComponent -
java.lang.RuntimeException: Assertion
  at org.intellij.samples.diagnostics.SampleLoggerComponent.initComponent
  at com.intellij.openapi.k.b.a.b(a.java:56)
  at com.intellij.openapi.k.b.a.getComponent (a.java:80)
```

В случае диагностики исключения, в файл попадает информация о версии IntelliJ IDEA, что может оказаться полезным, если ошибку обнаружил не сам разработчик, а конечный пользователь.

15.2.4. Сохранение и восстановление состояния компонент

Иногда компоненты нуждаются в сохранении своего состояния, чтобы при следующем запуске IntelliJ IDEA или при загрузке проекта они смогли восстановить его.

Для этих целей класс компоненты должен имплементировать интерфейс `JDOMExternalizable` или `NamedJDOMExternalizable` из пакета `com.intellij.openapi.util`.

15.2.4.1. Сохранение состояния компоненты

Процесс сохранения компонент состоит из следующих этапов:

1. Определение списка компонент, которые могут сохранять свое состояние на заданном уровне компонентной модели.
2. Определение для каждой из компонент имени файла, в котором они будут сохранять свое состояние.
3. Формирование XML-дерева для каждого из файлов посредством вызовов метода `writeExternal` для каждой из компонент. При этом в качестве параметра в данный метод передается отдельный элемент общего XML-дерева, в котором компонента может сохранять свое состояние. Если элементы относятся к одному и тому же файлу, они объединяются в единую XML-структуру.
4. Запись XML-структуры в соответствующие файлы.

Примечание

Вызов метода `writeExternal` носит не принудительный характер и просто сигнализирует о том, что компонента может сохранить свое состояние в заданном XML-элементе. Однако возможны случаи, когда имплементация этого метода игнорирует общую структуру и использует сугубо частную реализацию записи своего состояния, при этом компонента вправе сама решить, как данные записывать и где их сохранять.

Процесс сохранения происходит всякий раз, когда IntelliJ IDEA синхронизирует свою внутреннюю виртуальную файловую систему. Происходит это при любом изменении в настройках, при сохранении, при переключении на другое приложение, перед освобождением компоненты и т. д. Поэтому следует реализовывать метод `writeExternal` оптимальным образом.

В случае невозможности записи состояния данный метод должен генерировать исключение `WriteExternalException`.

15.2.4.2. Восстановление состояния компоненты

Процесс восстановления схож с процессом сохранения, однако запускается он не так часто. Состояние компоненты восстанавливается всего один раз — при инициализации компоненты, при этом сам процесс восстановления делится на следующие этапы:

1. Определение списка компонент, которые могут восстанавливать свое состояние на заданном уровне компонентной модели.
2. Определение для каждой из компонент имени файла, куда было записано состояние, если это было сделано стандартными средствами, а не собственной имплементацией.
3. Формирование XML-дерева для каждого из файлов.
4. Подразделение XML-дерева на элементы в соответствии с именами компонент.
5. Вызов метода `readExternal` с передачей ему в качестве параметра XML-элемента, соответствующего данной компоненте. При этом компонента должна прочитать атрибуты из переданного ей элемента или из вложенных в него элементов и, тем самым, восстановить свое состояние.

Примечание

Вызов метода `readExternal` носит не принудительный характер и просто сигнализирует о том, что компонента может восстановить свое состояние из заданного XML-элемента. Однако возможны случаи, когда имплементация этого метода игнорирует общую структуру и использует сугубо частную реализацию восстановления своего состояния, при этом компонента вправе сама решить, как и откуда считывать данные.

Данный метод в случае обнаружения неверных данных должен генерировать исключение `InvalidDataException`.

15.2.4.3. Реализация простого алгоритма сохранения и восстановления состояния компоненты

Существует специальный утилитный класс, с помощью которого вы можете легко сохранить и восстановить состояние вашей компоненты — `DefaultJDOMExternalizer`, не реализовывая сложную логику записи и чтения различных типов данных. С помощью данного класса можно сохранять и восстанавливать состояния всех примитивных типов данных, а также типа `String`, `Color` и `JDOMExternalizable`. Пример реализации компоненты с использованием данного утилитного класса представлен в листинге 15.8.

Листинг 15.8

```
package org.intellij.samples.externalizable;

import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.util.*;
import org.jdom.Element;

import java.awt.Color;

public class SampleApplicationComponent implements ApplicationComponent,
    NamedJDOMExternalizable {
    private static final String COMPONENT_NAME =
        "SampleApplicationComponent";
    private static final String FILE_NAME = "sample";
    private Externalizable mySettings;

    private static final class Externalizable
        implements JDOMExternalizable {
        public String MY_STRING_SETTING = "Default String";
        public int MY_INT_SETTING = 0;
        public boolean MY_BOOLEAN_SETTING = false;
        public Color MY_COLOR_SETTING = Color.BLUE;

        public void readExternal(Element element)
            throws InvalidDataException {
            DefaultJDOMExternalizer.readExternal(this, element);
        }

        public void writeExternal(Element element)
            throws WriteExternalException {
            DefaultJDOMExternalizer.writeExternal(this, element);
        }
    }

    public String getComponentName() {
        return COMPONENT_NAME;
    }

    public SampleApplicationComponent() {
```

```
        this.mySettings = new Externalizable();
    }

    public void initComponent() {}

    public void disposeComponent() {}

    public void readExternal(Element element)
        throws InvalidDataException {
        DefaultJDOMExternalizer.readExternal(mySettings, element);
    }

    public void writeExternal(Element element)
        throws WriteExternalException {
        DefaultJDOMExternalizer.writeExternal(mySettings, element);
    }

    public String getExternalFileName() {
        return FILE_NAME;
    }
}
```

Из данного примера видно, что для работы с `DefaultJDOMExternalizer` необходимо, чтобы сохраняемый им объект удовлетворял следующим требованиям:

- поля класса, которые необходимо сохранить, должны иметь модификатор доступа `puiblic`;
- класс должен реализовывать интерфейс `JDOMExternalizable`.

Результат сохранения состояния данной компоненты помещается в файл config/options/sample.xml и содержит значения полей объекта `mySettings` (листинг 15.9).

Листинг 15.9

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
  <component name="SampleApplicationComponent">
    <option name="MY_STRING_SETTING" value="Default String" />
    <option name="MY_INT_SETTING" value="0" />
```

```
    <option name="MY_BOOLEAN_SETTING" value="false" />
    <option name="MY_COLOR_SETTING" value="ff" />
  </component>
</application>
```

Обратите внимание, что значение атрибута `name` в элементе `component` соответствует имени компоненты, состояние которой представлено вложенными элементами. В данном случае, это результат работы `DefaultJDOMExternalizer`, который создал элементы `option`, где разместил информацию о полях класса `SampleApplicationComponent.Externalizable` (см. листинг 15.8).

15.2.4.4. Определение местоположения файла с состоянием компоненты

В момент сохранения или восстановления состояния IntelliJ IDEA определяет файл, в котором хранятся данные о состоянии компоненты (табл. 15.2).

Таблица 15.2. *Различия компонент при определении местоположения файла в стандартной реализации алгоритма сохранения и восстановления*

Уровень	Возможность задания файла в стандартной реализации	Местоположение файла, определяемое IntelliJ IDEA
Компоненты приложения	Имя файла можно определить, имплементировав интерфейс NamedJDOMExternalizable	Файл будет располагаться в поддиректории options конфигурационной директории IntelliJ IDEA. По умолчанию — это файл other.xml. Если реализован интерфейс NamedJDOMExternalizable, то имя файла соответствует указанному имени в имплементации
Проектные компоненты	Можно определить, что компонента должна сохранять свое состояние в файле проектного окружения (iws), а не в проектном файле (ipr). Для этого необходимо сделать указание в XML-дескрипторе плагина	Состояние будет сохранено в проектном файле (ipr), либо в файле проектного окружения (iws), в соответствии с настройками в XML-дескрипторе плагина
Модульные компоненты	Нет	Состояние сохраняется только в файле модуля (iml)

Для задания местоположения файла проектной компоненты необходимо в XML-дескрипторе плагина указать дополнительную настройку (листинг 15.10).

Листинг 15.10

```
<project-components>
    <component>
        <implementation-class>...</implementation-class>
        <option name="workspace" value="true"/>
    </component>
</project-components>
```

Примечание

В IntelliJ IDEA присутствует один загруженный шаблон проекта, который содержит в себе базовые настройки. У него нет проектного окружения, и свое состояние этот шаблон-проект всегда, не зависимо от настроек в XML-дескрипторе плагина, будет сохранять в файле options/project.default.xml относительно директории с настройками IntelliJ IDEA.

15.2.5. Система действий в IntelliJ IDEA

Действием называется имплементация абстрактного класса `AnAction` из пакета `com.intellij.openapi.actionSystem`. Действия, объединенные в группу, могут образовывать пункты меню или панели инструментов. В группу действий могут входить также и подгруппы (например, в меню).

Действия могут быть *простыми* и *переключаемыми*. Первый тип действий используется для сигнализации о команде от пользователя сделать что-либо, а второй — для включения или выключения каких-либо процессов. Если нужно создать действие-переключатель, необходимо имплементировать абстрактный класс `ToggleAction`, который расширяет класс `AnAction`.

15.2.5.1. Простые действия

Простые действия — это действия, которые вызываются по команде пользователя. Например, команда пользователя открыть файл приведет к выполнению действия **Open File** (Открыть файл) (*см. раздел 3.2.1.4*) в меню **File** (Файл).

В момент выполнения команды происходит событие, которое передается в метод `actionPerformed` вызываемого действия. Вместе с событием передается контекстная информация (для получения которой следует использовать метод `getDataContext`). Контекстная информация содержит указание на текущий проект, выделенный файл и другую полезную информацию (листинг 15.11).

Листинг 15.11

```
package org.intellij.samples.actionSystem;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.ui.Messages;

public class SampleAction extends AnAction {
    public SampleAction() {
        super("Sample Action");
    }

    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        Messages.showInfoMessage(
            project, "Action performed", "Sample Action"
        );
    }
}
```

Для изменения визуального представления действия в объекте-событии (`AnActionEvent`) также передается специальный объект, который можно получить методом `getPresentation`.

Примечание

Для изменения визуального оформления действия рекомендуется перегружать метод `update`. При этом следует помнить о том, что данный метод вызывается системой периодически и многократно, поэтому реализовать данный метод необходимо с расчетом на максимальное быстродействие.

15.2.5.2. Переключаемые действия

Переключаемые действия — это действия, которые служат для переключения состояний, режимов и других переключений в системе. Примером переключаемого действия может служить переключатель **Show Whitespaces** (Показывать

пробелы) (*см. раздел 3.2.4.3*) из меню **View** (Просмотр) или пиктограмма **Show Members** (Показывать члены класса) в инструментальном окне **Project** (Проект) (*см. раздел 3.4.8.2*).

Для переключаемых действий необходимо имплементировать методы `isSelected` и `setSelected`. Эти методы определяют состояние "включено/выключено".

Метод `isSelected` необходим для того, чтобы система могла определить состояние переключателя в любой момент времени. Метод `setSelected` вызывается для принудительной установки состояния переключателя.

Внимание

Методы `isSelected` и `update` вызываются системой периодически и многократно. Данные методы должны быть реализованы с расчетом на максимальное быстродействие.

Метод `setSelected` является аналогом метода `actionPerformed`. В параметрах данного метода также присутствует событие, сгенерированное системой в момент команды пользователя в ответ на изменение состояния. Вторым параметром данного метода передается *желаемое* состояние переключателя. Однако это совсем не значит, что переключатель после этого события изменит свое состояние: действие может само определить, менять свое состояние или нет (листинг 15.12).

Листинг 15.12

```
package org.intellij.samples.actionSystem;

import com.intellij.openapi.actionSystem.*;
import com.intellij.openapi.util.IconLoader;

import javax.swing.*;
import java.awt.event.InputEvent;
import java.awt.event.MouseEvent;

public class SampleToggleAction extends ToggleAction {
    private boolean selected;

    public SampleToggleAction() {
        super(
            "Sample Toggle Action",
```

```
            "Sample of toggle action",
            IconLoader.getIcon("/general/template.png")
        );
    }

    public boolean isSelected(AnActionEvent e) {
        return selected;
    }

    public void setSelected(AnActionEvent e, boolean state) {
        InputEvent inputEvent = e.getInputEvent();
        if (inputEvent instanceof MouseEvent) {
            MouseEvent event = (MouseEvent)inputEvent;
            DefaultActionGroup group = new DefaultActionGroup(
                "Sample Popup", true
            );
            group.add(
                new ToggleAction(Boolean.TRUE.toString()) {
                    public boolean isSelected(AnActionEvent e) {
                        return selected;
                    }

                    public void setSelected(AnActionEvent e, boolean s) {
                        selected = true;
                    }
                }
            );
            group.add(
                new ToggleAction(Boolean.FALSE.toString()) {
                    public boolean isSelected(AnActionEvent e) {
                        return !selected;
                    }

                    public void setSelected(AnActionEvent e, boolean s) {
                        selected = false;
                    }
                }
```

```
            );
            ActionManager actionManager = ActionManager.getInstance();
            ActionPopupMenu menu = actionManager.createActionPopupMenu(
                ActionPlaces.UNKNOWN, group
            );
            JPopupMenu popupMenu = menu.getComponent();
            popupMenu.pack();
            popupMenu.show(
                event.getComponent(),
                event.getX(),
                event.getY()
            );
        }
        selected = state;
    }
}
```

15.2.5.3. Создание всплывающего меню

Для создания всплывающего меню необходимо воспользоваться классом `ActionManager` из пакета `com.intellij.openapi.actionSystem`. Следует подготовить группу действий и передать ее в метод `createActionPopupMenu`, который вернет объект типа `ActionPopupMenu`.

Класс `ActionPopupMenu` является оберткой класса `JPopupMenu` из библиотеки Swing. Вы можете получить Swing-объект, вызвав метод `getComponent`.

Создание такого меню может пригодиться в редакторе или при реализации особого типа действия панели инструментов, подобного действию **Add** (Добавить) в диалоговом окне **Application Servers** (Серверы приложений) (*см. раздел 4.1.20*) (если в IntelliJ IDEA установлены серверы WebLogic и Tomcat, то нажатие на эту кнопку выведет всплывающее меню, в котором будет предложено выбрать тип сервера приложений). Реализация такого функционала рассмотрена в предыдущем разделе (см. листинг 15.12).

15.2.5.4. Создание панели инструментов

Панель инструментов — это набор действий, отображаемых на одной визуальной панели в виде пиктограмм. Панель инструментов зачастую дублирует набор действий, которые могут быть вызваны непосредственно из меню или комбинацией клавиш.

Для создания панели инструментов необходимо воспользоваться классом `ActionManager` из пакета `com.intellij.openapi.actionSystem`. Следует под-

готовить группу действий и передать ее в метод `createActionToolbar`, который вернет объект типа `ActionToolbar`.

Класс `ActionToolbar` является простой оберткой класса `JComponent` из библиотеки Swing. Вы можете получить Swing-объект, вызвав метод `getComponent` (см. листинг 15.19).

15.2.5.5. Модификация существующих групп действий

При необходимости можно изменить любую доступную системную группу действий. Для этого в OpenAPI IntelliJ IDEA предусмотрен механизм регистрации действий. При регистрации действия или группы действий назначается идентификатор, с помощью которого можно обратиться к данному объекту из любой компоненты IntelliJ IDEA, насколько это позволяет загрузчик классов.

Большинство стандартных групп действий и сами действия зарегистрированы, поэтому разработчик может получить данные объекты и менять их, т. е. добавлять свои пункты меню или свои пиктограммы на стандартную панель инструментов (листинг 15.13).

Для получения зарегистрированного действия необходимо указать идентификатор. Все стандартные идентификаторы указаны в интерфейсе `IdeActions` из пакета `com.intellij.openapi.actionSystem`.

Добавлять действия в группу действий можно, только если эта группа представлена классом `DefaultActionGroup`, так как абстрактный класс `ActionGroup` не позволяет изменять группу действий. Поэтому при получении группы необходимо убедиться, что это объект класса `DefaultActionGroup`, его расширение или частная имплементация, позволяющая менять группу.

Листинг 15.13

```
package org.intellij.samples.actionSystem;

import com.intellij.openapi.actionSystem.ActionManager;
import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.DefaultActionGroup;
import com.intellij.openapi.actionSystem.IdeActions;
import com.intellij.openapi.components.ApplicationComponent;

public class SampleActionManager implements ApplicationComponent {
    private static final String NAME = "SampleActionManager";
    private static final String GROUP_SAMPLE_MENU = "SampleMenuGroup";
```

```
    private static final String ACTION_SAMPLE_TOGGLE =
        "SampleToogleAction";

    public String getComponentName() {
        return NAME;
    }

    public void initComponent() {
        ActionManager actionManager = ActionManager.getInstance();
        AnAction mainMenu = actionManager.getAction(
            IdeActions.GROUP_MAIN_MENU
        );
        if (mainMenu instanceof DefaultActionGroup) {
            DefaultActionGroup mmGroup = (DefaultActionGroup)mainMenu;
            DefaultActionGroup sampleGroup = new DefaultActionGroup(
                "Sample", false
            );
            sampleGroup.add(new SampleAction());

            // Register action to enable removing in dispose
            actionManager.registerAction(GROUP_SAMPLE_MENU, sampleGroup);

            // Append action in menu
            mmGroup.add(sampleGroup);
        }

        AnAction mainToolbar = actionManager.getAction(
            IdeActions.GROUP_MAIN_TOOLBAR
        );
        if (mainToolbar instanceof DefaultActionGroup) {
            DefaultActionGroup mtGroup = (DefaultActionGroup)mainToolbar;
            SampleToggleAction action = new SampleToggleAction();

            // Register action to enable removing
            actionManager.registerAction(ACTION_SAMPLE_TOGGLE, action);

            mtGroup.add(action);
        }
    }

    public void disposeComponent() {
```

```
        ActionManager actionManager = ActionManager.getInstance();
        AnAction mainMenu = actionManager.getAction(
            IdeActions.GROUP_MAIN_MENU
        );
        if (mainMenu instanceof DefaultActionGroup) {
            DefaultActionGroup mmGroup = (DefaultActionGroup)mainMenu;
            AnAction sampleGroup = actionManager.getAction(
                GROUP_SAMPLE_MENU
            );
            if (sampleGroup != null) {
                // Deregister action
                actionManager.unregisterAction(GROUP_SAMPLE_MENU);

                // Remove action from menu
                mmGroup.remove(sampleGroup);
            }
        }

        AnAction mainToolbar = actionManager.getAction(
            IdeActions.GROUP_MAIN_TOOLBAR
        );
        if (mainToolbar instanceof DefaultActionGroup) {
            DefaultActionGroup mtGroup = (DefaultActionGroup)mainToolbar;
            AnAction sampleToggle = actionManager.getAction(
                ACTION_SAMPLE_TOGGLE
            );
            if (sampleToggle != null) {
                // Deregister action
                actionManager.unregisterAction(ACTION_SAMPLE_TOGGLE);

                // Remove action from menu
                mtGroup.remove(sampleToggle);
            }
        }
    }
}
```

15.2.6. Стандартные визуальные компоненты IntelliJ IDEA

В составе библиотеки OpenAPI есть классы для построения визуальных компонент IntelliJ IDEA, таких как диалоговые окна, панели инструментов, инструментальные окна, а также ряд служебных компонент.

15.2.6.1. Панели конфигурации

В IntelliJ IDEA существует система настройки компонент. Панели конфигурации компонент располагаются в диалоге настроек **IDE Settings** (Настройки IDE) (<Ctrl>+<Alt>+<S>) (*см. раздел 4.1*).

Чтобы добавить панель для конфигурации плагина, необходимо имплементировать интерфейс `Configurable` из пакета `com.intellij.openapi.options` (листинг 15.14). Кроме того, данная имплементация должна быть либо компонентой приложения для хранения данных приложения, либо проектной компонентой для хранения данных проекта.

Листинг 15.14

```
package org.intellij.samples.ui.options;

import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.options.Configurable;
import com.intellij.openapi.options.ConfigurationException;

import javax.swing.*;

public class SampleConfigurable implements Configurable,
    ApplicationComponent {
    private UIForm uiForm;

    public String getDisplayName() {
        return "Sample";
    }

    public Icon getIcon() {
        return null;
    }

    public String getHelpTopic() {
```

```
        return null;
    }

    public JComponent createComponent() {
        if (uiForm == null) {
            uiForm = new UIForm();
        }
        return uiForm.uiComponent;
    }

    public boolean isModified() {
        SampleConfiguration configuration =
            SampleConfiguration.getInstance();
        String text = uiForm.uiString.getText();
        return !text.equals(configuration.getString());
    }

    public void apply() throws ConfigurationException {
        SampleConfiguration configuration =
            SampleConfiguration.getInstance();
        configuration.setString(uiForm.uiString.getText());
    }

    public void reset() {
        SampleConfiguration configuration =
            SampleConfiguration.getInstance();
        uiForm.uiString.setText(configuration.getString());
    }

    public void disposeUIResources() {
    }

    public String getComponentName() {
        return "SampleConfigurable";
    }

    public void initComponent() {
```

```
    }

    public void disposeComponent() {
    }

    private static final class UIForm {
        private JPanel uiComponent;
        private JTextField uiString;
    }
}
```

Примечание

В данном примере используется форма, построенная с помощью визуального редактора форм. Поэтому класс `UIForm` содержит только поля, а инициализация и разметка выполнены автоматически с помощью визуального редактора форм (*см. главу 7*).

Обратите внимание, что объект `uiForm` создается в методе `createComponent`. Это сделано специально, чтобы создаваемые компоненты наследовали визуальный стиль IntelliJ IDEA. Если бы объект `uiForm` инициализировался в конструкторе, то оформление графических элементов было бы наследовано от стандартного оформления (Look&Feel) Swing.

Реализация конфигурационной панели напрямую связана с сохранением и восстановлением состояний компонент (*см. раздел 15.2.4*). При вызове метода `apply`, данные из класса формы должны быть скопированы в компоненту, которая отвечает за сохранение данных. При вызове метода `reset` данные, наоборот, должны быть загружены из компоненты в визуальную форму. Метод `isModified` определяет, поменялись данные или нет.

Если пользователь сделал изменения в визуальных компонентах конфигурационной панели, то необходимо проверить, нужно ли применять данные изменения или нет. Кнопка **Apply** (Применить) меняет свое состояние в соответствии с методом `isModified`. Реализация компоненты, которая отвечает за сохранение и восстановление данных, представлена в листинге 15.15.

Листинг 15.15

```
package org.intellij.samples.ui.options;

import com.intellij.openapi.application.Application;
import com.intellij.openapi.application.ApplicationManager;
```

```
import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.util.DefaultJDOMExternalizer;
import com.intellij.openapi.util.InvalidDataException;
import com.intellij.openapi.util.JDOMExternalizable;
import com.intellij.openapi.util.WriteExternalException;
import org.jdom.Element;

public class SampleConfiguration implements JDOMExternalizable,
    ApplicationComponent {
    private Externalizable myExternalizable;

    public SampleConfiguration() {
        myExternalizable = new Externalizable();
    }

    private static final class Externalizable
        implements JDOMExternalizable {
        public String MY_STRING = "Hello World!";

        public void readExternal(Element element)
            throws InvalidDataException {
            DefaultJDOMExternalizer.readExternal(this, element);
        }

        public void writeExternal(Element element)
            throws WriteExternalException {
            DefaultJDOMExternalizer.writeExternal(this, element);
        }
    }

    public void readExternal(Element element)
        throws InvalidDataException {
        myExternalizable.readExternal(element);
    }

    public void writeExternal(Element element)
        throws WriteExternalException {
```

```
        myExternalizable.writeExternal(element);
    }

    public String getComponentName() {
        return "SampleConfiguration";
    }

    public String getString() {
        return myExternalizable.MY_STRING;
    }

    public void setString(String value) {
        myExternalizable.MY_STRING = value;
    }

    public void initComponent() {
    }

    public void disposeComponent() {
    }

    public static final SampleConfiguration getInstance() {
        Application application = ApplicationManager.getApplication();
        return (SampleConfiguration)application.getComponent(
            SampleConfiguration.class
        );
    }
}
```

15.2.6.2. Диалоговые окна IntelliJ IDEA

Диалоговые окна в IntelliJ IDEA построены на основе класса `DialogWrapper` из пакета `com.intellij.openapi.ui` (см. листинг 15.16). Данный класс представляет собой обертку класса `JDialog` из библиотеки Swing. Внутреннее наполнение визуальными компонентами осуществляется с помощью `BorderLayout`, поэтому существует понятие *северной панели*, *центральной панели* и *южной панели*. В простейшей реализации разработчик обязан определить только центральную панель, которая создается в методе `createCenterPanel`, однако он также может перегрузить методы

createNorthPanel и createSouthPanel, чтобы определить северную и южную панели диалогового окна. По умолчанию в южной панели располагаются кнопки диалогового окна, которые могут также быть переопределены разработчиком.

Существует еще одно деление на области — *заголовочная панель*, которая располагается над северной панелью, и *объединенная панель*, которая в базовой реализации содержит северную, центральную и южную панели. Заголовочная панель определяется перегрузкой метода createTitlePanel, а объединенная панель создается в методе createContentPane. При желании все эти методы можно реализовать частным образом по усмотрению разработчика. Однако следует помнить, что создание объединенной панели — это лишь создание контейнера, куда будут помещены северная, центральная и южная панели.

Кнопки диалога, команды или действия задаются с помощью перегрузки метода createActions, который должен возвратить массив объектов Action из библиотеки Swing. По умолчанию этот массив содержит два стандартных действия: OK и Cancel.

Данный метод определяет действия, по которым в стандартной реализации будут созданы кнопки в правой части южной панели диалогового окна. Для определения действий для кнопок, которые могут располагаться в левой части панели, используйте метод createLeftSideActions. Этот метод по умолчанию не возвращает ни одного действия.

Стандартный алгоритм реализации южной панели преобразует массив, возвращенный методом createActions или createLeftSideActions, в кнопки диалогового окна. При преобразовании вызывается метод createJButtonForAction, который также может быть переопределен разработчиком. Метод получает на вход объект типа Action, а возвращает уже готовый к помещению на южную панель объект типа JButton.

Листинг 15.16

```
package org.intellij.samples.ui.dialog;

import com.intellij.openapi.ui.DialogWrapper;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.util.IconLoader;
import javax.swing.*;

public class SampleDialog extends DialogWrapper {
    public SampleDialog(Project project) {
```

```
        super(project, false);
        init();
    }

    protected JComponent createCenterPanel() {
        return new JLabel(IconLoader.getIcon("/idea_about.png"));
    }
}
```

Построение визуальных компонент диалоговых окон можно подразделить на следующие этапы:

- создание заголовочной панели методом `createTitlePanel`;
- создание объединенной панели методом `createContentPane`;
- создание северной панели методом `createNorthPanel`;
- создание центральной панели методом `createCenterPanel`;
- создание южной панели методом `createSouthPanel`;
  - создание действий методом `createActions`;
  - преобразование действий в кнопки методом `createJButtonForAction`;
  - добавление кнопок на южную панель;
- добавление северной, центральной и южной панелей на объединенную панель.

Для создания визуальных компонент необходимо вызвать метод `init`, который выполнит построение визуальных компонент и подготовит диалоговое окно для показа.

Диалоговые окна имеют возможность сохранять свои размеры и местоположение с помощью компоненты `DimensionService`. Чтобы разрешить диалоговому окну сохранять местоположение и размеры, необходимо перегрузить метод `getDimensionServiceKey`, который вернет идентификатор для компоненты `DimensionService`. По этому идентификатору компонента при закрытии диалогового окна сохранит размеры и положение, чтобы при следующем открытии диалогового окна восстановить их.

Вызов диалогового окна показан в листинге 15.17.

Листинг 15.17

```
package org.intellij.samples.ui.actions;

import com.intellij.openapi.actionSystem.AnAction;
```

```
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.actionSystem.DataContext;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.ui.DialogWrapper;
import com.intellij.openapi.ui.Messages;
import org.intellij.samples.ui.dialog.SampleDialog;

public class ShowSampleDialogAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        DataContext dataContext = e.getDataContext();
        Project project = (Project)dataContext.getData(
                DataConstants.PROJECT
        );
        SampleDialog dialog = new SampleDialog(project);
        dialog.setTitle("Sample Dialog");
        dialog.show();
        switch (dialog.getExitCode()) {
            case DialogWrapper.OK_EXIT_CODE:
                Messages.showInfoMessage(
                        project, "OK pressed", "Sample Dialog"
                );
                break;
            case DialogWrapper.CANCEL_EXIT_CODE:
                Messages.showInfoMessage(
                        project, "Cancel pressed", "Sample Dialog"
                );
                break;
        }
    }
}
```

Примечание

В данном примере использован класс `Messages` из пакета `com.intellij.openapi.ui`, который можно применять для отображения простых информационных диалоговых окон, например, при сообщении пользователю об ошибке ввода данных.

15.2.6.3. Создание инструментальных окон (ToolWindow)

Инструментальные окна в IntelliJ IDEA представлены следующими компонентами (*см. раздел 3.4*):

- проектный навигатор (**Project** (Проект));
- инструментальное окно **Ant**;
- файловый навигатор (окно **Commander** (Коммандер));
- инструментальное окно **TODO** (Заметки);
- другие.

Все эти визуальные компоненты зарегистрированы в системе с уникальными идентификаторами, которые также являются их названием в строке вызова инструментальных окон. Все стандартные зарегистрированные идентификаторы представлены в интерфейсе `ToolWindowId` из пакета `com.intellij.openapi.wm`.

Создание инструментального окна представляет собой регистрацию его в системе с помощью класса `ToolWindowManager` из пакета `com.intellij.openapi.wm`. Удаление инструментального окна из системы производится также через этот класс. При регистрации необходимо передать в метод `registerToolWindow` идентификатор инструментального окна (наименование), реализацию `JComponent` и местоположение. Местоположение задается классом констант `ToolWindowAnchor`:

- `TOP` — инструментальное окно располагается в верхней части окна IntelliJ IDEA;
- `LEFT` — инструментальное окно располагается в левой части окна IntelliJ IDEA;
- `BOTTOM` — инструментальное окно располагается в нижней части окна IntelliJ IDEA;
- `RIGHT` — инструментальное окно располагается в правой части окна IntelliJ IDEA.

После создания (регистрации) инструментального окна можно поменять некоторые параметры в полученном объекте `ToolWindow`. Например, можно установить заголовок, пиктограмму, видимость и доступность (листинг 15.18).

Кроме того, в системе происходит автоматическое создание действия, которое отвечает за активацию созданного инструментального окна. Действие помещается в меню **Window** (Окно) (*см. раздел 3.2.12*) и имеет идентификатор, который строится на основании идентификатора инструментального окна. Если идентификатор содержит пробелы, то они удаляются при формировании идентификатора действия. Идентификатор формируется путем прибавления префикса `Activate` и суффикса `ToolWindow`.

Примечание

Данная особенность системы не является официальной. Ее использование может привести к несовместимости в будущих версиях, так как разработчики могут изменить алгоритм формирования идентификатора для активационного действия. Поэтому следует делать проверку существования действия с необходимым идентификатором в системе.

Зная такое поведение системы, можно сразу после регистрации инструментального окна получить из системы соответствующее активационное действие и назначить ему комбинацию клавиш, как это сделано почти для всех стандартных инструментальных окон.

Примечание

Рекомендуется использовать системный формат комбинаций клавиш с применением клавиши <Alt>. В этом случае в инструментальных окнах показываются комбинации клавиш для активации. Например, для инструментального окна **Project** (Проект) — это <Alt>+<1>, для **Structure** (Структура) — это <Alt>+<7>. Если для активационного действия назначается комбинация клавиш в другом формате, она не будет отображаться в инструментальных окнах.

Листинг 15.18

```
package org.intellij.samples.ui.actions;

import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.actionSystem.KeyboardShortcut;
import com.intellij.openapi.actionSystem.ToggleAction;
import com.intellij.openapi.keymap.KeymapManager;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.util.IconLoader;
import com.intellij.openapi.wm.ToolWindow;
import com.intellij.openapi.wm.ToolWindowAnchor;
import com.intellij.openapi.wm.ToolWindowManager;
import org.intellij.samples.ui.toolWindow.SampleToolWindow;

import javax.swing.*;
import java.awt.event.InputEvent;
import java.awt.event.KeyEvent;

public class ToggleSampleToolWindowAction extends ToggleAction {
```

```
    private static final String SAMPLE_TOOLWINDOW_ID = "Sample";
    private static final String SAMPLE_TOOLWINDOW_ACTION_ID =
        "Activate" +
        SAMPLE_TOOLWINDOW_ID.replaceAll(" ", "") +
        "ToolWindow";

    public boolean isSelected(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        if (project != null) {
            ToolWindowManager manager = ToolWindowManager.getInstance(
                project
            );
            ToolWindow toolWindow = manager.getToolWindow(
                SAMPLE_TOOLWINDOW_ID
            );
            return toolWindow != null && toolWindow.isAvailable();
        }
        return false;
    }

    public void update(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        e.getPresentation().setEnabled(project != null);
    }

    public void setSelected(AnActionEvent e, boolean state) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        if (project != null) {
            ToolWindowManager manager = ToolWindowManager.getInstance(
                project
            );
```

```
            ToolWindow toolWindow = manager.getToolWindow(
                SAMPLE_TOOLWINDOW_ID
            );
            if (state) {
                if (toolWindow == null) {
                    toolWindow = manager.registerToolWindow(
                        SAMPLE_TOOLWINDOW_ID,
                        new SampleToolWindow(),
                        ToolWindowAnchor.BOTTOM
                    );

                    KeymapManager kmManager =
                        KeymapManager.getInstance();
                    kmManager.getActiveKeymap().addShortcut(
                        SAMPLE_TOOLWINDOW_ACTION_ID,
                        new KeyboardShortcut(
                            KeyStroke.getKeyStroke(
                                KeyEvent.VK_9, InputEvent.ALT_MASK
                            ),
                            null
                        )
                    );
                    toolWindow.setTitle("Title");
                    toolWindow.setIcon(
                        IconLoader.getIcon("/general/template.png")
                    );
                }
                toolWindow.setAvailable(true, null);
            } else {
                if (toolWindow != null) {
                    toolWindow.setAvailable(false, null);
                }
            }
        }
    }
}
```

Внешний вид инструментального окна формируется из установленных атрибутов. Для удобства элементы пронумерованы (рис. 15.7):

- наименование (идентификатор) на рисунке помечено номером **1**;
- заголовок окна — номером **2**;
- пиктограмма — номером **3**;
- комбинация клавиш для активации инструментального окна видна в области под номером **4**.

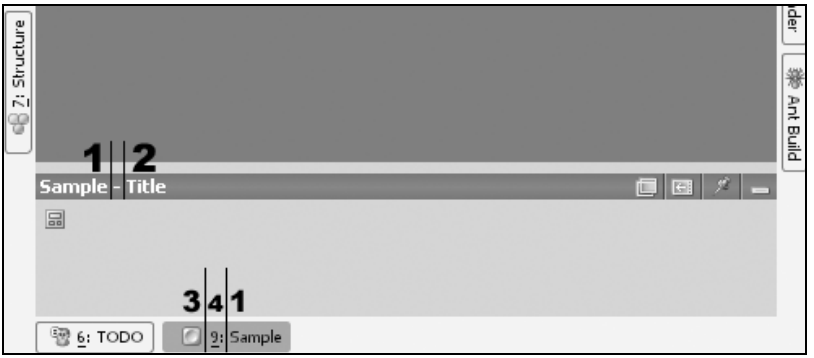


Рис. 15.7. Расположение элементов инструментального окна

В основной области инструментального окна располагается компонента `JComponent`, переданная в метод `registerToolWindow`. Ее реализация может сама определять внутреннее размещение визуальных компонент (листинг 15.19).

Листинг 15.19

```
package org.intellij.samples.ui.toolWindow;

import com.intellij.openapi.actionSystem.*;

import javax.swing.*;
import java.awt.*;

public class SampleToolWindow extends JPanel {
    public SampleToolWindow() {
        super(new BorderLayout());

        DefaultActionGroup group = new DefaultActionGroup();
```

```
        ActionManager actionManager = ActionManager.getInstance();
        AnAction action = actionManager.getAction("ShowSampleDialog");
        if (action != null) {
            group.add(action);
        }
        action = actionManager.getAction("ShowSampleFileChooser");
        if (action != null) {
            group.add(action);
        }
        if (group.getChildrenCount() > 0) {
            ActionToolbar toolbar = actionManager.createActionToolbar(
                ActionPlaces.UNKNOWN, group, true
            );
            add(toolbar.getComponent(), BorderLayout.NORTH);
        }
    }
}
```

15.2.6.4. Компонента для выбора файлов и директорий

Иногда требуется, чтобы пользователь определил путь к файлу или к директории. Проще всего это сделать визуально. В IntelliJ IDEA существует стандартная компонента, которая предоставляет удобный интерфейс для поиска и выбора файлов и директорий.

Данная компонента представлен в библиотеке OpenAPI классом `FileChooser` из пакета `com.intellij.openapi.fileChooser`, который содержит статический метод `chooseFiles` с различными вариациями в наборе параметров. Для вызова данного метода необходимо указать дескриптор, который задает поведение диалогового окна выбора файлов и директорий (листинг 15.20).

Листинг 15.20

```
package org.intellij.samples.ui.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
```

```
import com.intellij.openapi.fileChooser.FileChooser;
import com.intellij.openapi.fileChooser.FileChooserDescriptor;
import com.intellij.openapi.fileChooser.FileChooserDescriptorFactory;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.ui.Messages;
import com.intellij.openapi.vfs.VirtualFile;

public class ShowSampleFileChooser extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        // Browse for folder
        FileChooserDescriptor descriptor =
            FileChooserDescriptorFactory.createSingleFolderDescriptor();
        descriptor.setTitle("Choose folder");
        descriptor.setDescription("Please, choose single folder");
        VirtualFile[] virtualFiles = FileChooser.chooseFiles(
            project,
            descriptor
        );
        if (virtualFiles.length > 0) {
            Messages.showInfoMessage(
                virtualFiles[0].getPath(), "Selected folder"
            );
        }
    }
}
```

Для создания экземпляра класса `FileChooserDescriptor` из пакета `com.intellij.openapi.fileChooser` в данном примере используется класс `FileChooserDescriptorFactory` из того же пакета. Данный класс может создавать дескрипторы для различных случаев использования компоненты выбора файлов и директорий. После создания дескриптора можно также изменить стандартные свойства, такие как заголовок (метод `setTitle`) и описание (метод `setDescription`).

15.2.6.5. Визуальные компоненты общего назначения

В OpenAPI IntelliJ IDEA существуют дополнительные визуальные компоненты общего назначения. Например, класс `Messages` из пакета `com.intellij.openapi.ui` используется для вывода информационных сообщений пользователю. Пример использования данного класса приведен в листинге 15.20.

Кроме того, в пакете `com.intellij.openapi.ui` содержатся компоненты, с помощью которых можно создавать визуальные формы. Вот некоторые из этих компонент:

- `TextFieldWithBrowseButton`. Данная компонента позволяет получить текстовое поле с кнопкой для выбора. Эта компонента расширяет класс `ComponentWithBrowseButton`, который также может быть использован аналогично текстовому полю;
- `Splitter`. Компонента для разделения области визуальных компонент. Эта компонента используется, в частности, для разделения редактора IntelliJ IDEA.

Примечание

Для размещения данных компонент на панели инструментов визуального редактора IntelliJ IDEA вам необходимо включить библиотеку Trove4J в число проектных библиотек. Подробнее о настройке визуального редактора *см. раздел 7.3*.

15.2.7. Редактор IntelliJ IDEA

Редактор IntelliJ IDEA, это, пожалуй, самая лучшая часть во всей IntelliJ IDEA. Работа с ним на более низком уровне доступна через библиотеку OpenAPI. Разработчики плагинов могут создавать на базе редактора IntelliJ IDEA свои собственные редакторы.

В качестве примера такой реализации можно привести плагин PropertiesEditor, который предоставляет возможность "умного" редактирования файлов свойств (с расширением properties) с возможностями подсветки синтаксиса, рефакторинга и поиска.

Различают файловый редактор и текстовый редактор. Файловый редактор предоставляет разработчику самому реализовать редактор для заданного типа файлов (на этом принципе построены плагины, которые делают возможным просмотр графических файлов или сформированных изображений и диаграмм внутри IntelliJ IDEA). Текстовый редактор — это редактор, который уже написали разработчики IntelliJ IDEA, и его можно использовать с минимальными усилиями, изменять подсветку синтаксиса, добавлять и удалять данные и т. п. При этом внешне текстовый редактор выглядит точно так же, как и стандартный редактор текстовых файлов.

15.2.7.1. Создание редактора файлов

Библиотека OpenAPI позволяет создавать собственные реализации файловых редакторов. В данный момент в IntelliJ IDEA существуют следующие стандартные редакторы:

- редактор Java;
- редактор XML;
- редактор Ant-скрипта, построенный на базе XML-редактора;
- редактор HTML;
- редактор JSP (комбинированный редактор JSP/HTML/XML/Java);
- редактор визуальных форм (*см. главу 7*);
- стандартный редактор по типам файлов (JavaScript, IDL). Данный тип редактора подсвечивает синтаксис с учетом настроек для заданного типа файла;
- стандартный текстовый файл для всех текстовых файлов без подсветки синтаксиса.

Остальные файлы, в том числе изображения и пиктограммы, в IntelliJ IDEA не могут быть открыты. Поэтому существуют плагины, которые добавляют для различных типов файлов свои собственные редакторы.

Для создания собственного редактора необходимо имплементировать интерфейс `FileEditorProvider` из пакета `com.intellij.openapi.fileEditor`. Эта имплементация должна также быть компонентой приложения (*см. раздел 15.2.2.1*), иначе IntelliJ IDEA не узнает о ее существовании (листинг 15.21).

В методе `accept` должно приниматься решение о том, будет ли файл открываться с помощью редактора, определенного в этой реализации. А метод `createEditor` должен создать необходимый редактор после того, как метод `accept` вернет положительный ответ.

Листинг 15.21

```
package org.intellij.samples.editor.fileEditor;

import com.intellij.openapi.components.ApplicationComponent;
import com.intellij.openapi.fileEditor.FileEditor;
import com.intellij.openapi.fileEditor.FileEditorPolicy;
import com.intellij.openapi.fileEditor.FileEditorProvider;
import com.intellij.openapi.fileEditor.FileEditorState;
import com.intellij.openapi.project.Project;
```

```
import com.intellij.openapi.vfs.VirtualFile;
import org.intellij.plus.fileTypes.FileTypeDescriptor;
import org.intellij.plus.fileTypes.FileTypeDescriptorManager;
import org.jdom.Element;

import javax.swing.*;
import java.awt.*;
import java.io.IOException;

public class SampleFileEditorProvider implements FileEditorProvider,
    ApplicationComponent {
    private static final String NAME = "SampleFileEditorProvider";
    private static final String FILE_EDITOR_ID = "SampleFileEditor";

    public boolean accept(Project project, VirtualFile file) {
        // We will open only small images
        // less that 1MB
        if (file.getLength() <= (1L << 20)) {
            try {
                byte[] bytes = file.physicalContentsToByteArray();
                Toolkit defaultToolkit = Toolkit.getDefaultToolkit();
                Image image = defaultToolkit.createImage(bytes);
                return image != null;
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
        return false;
    }

    public FileEditor createEditor(Project project, VirtualFile file) {
        try {
            byte[] bytes = file.physicalContentsToByteArray();
            Icon icon = new ImageIcon(bytes);
            return new SampleFileEditor(file.getName(), icon);
        } catch (IOException e) {
            throw new RuntimeException(e);
```

```
        }
    }

    public void disposeEditor(FileEditor editor) {
    }

    public FileEditorState readState(
        Element sourceElement,
        Project project,
        VirtualFile file
        ) {
        return new FileEditorState() {
        };
    }

    public void writeState(
        FileEditorState state,
        Project project,
        Element targetElement
        ) {
    }

    public String getEditorTypeId() {
        return FILE_EDITOR_ID;
    }

    public FileEditorPolicy getPolicy() {
        return FileEditorPolicy.NONE;
    }

    public String getComponentName() {
        return NAME;
    }

    public void initComponent() {
        FileTypeDescriptorManager manager =
            FileTypeDescriptorManager.getInstance();
```

```
        FileTypeDescriptor descriptor = new FileTypeDescriptor();
        descriptor.setBinary(true);
        descriptor.setDefaultExtension("png");
        descriptor.addExtension("png");
        descriptor.addExtension("gif");
        descriptor.addExtension("jpeg");
        descriptor.addExtension("jpg");
        descriptor.setName("Graphics");
        descriptor.setDescription("Graphics files");
        descriptor.setIconPath(
            "/org/intellij/samples/editor/fileEditor/images.png"
        );
        manager.registerFileType(descriptor);
    }

    public void disposeComponent() {
    }
}
```

Примечание

В данном примере используется библиотека PLUS для регистрации типов файлов. Библиотека доступна на сайте сообщества разработчиков по адресу **http://www.intellij.org/twiki/bin/view/Main/PlusLibrary**.

После того как редактор будет закрыт, будет вызван метод `disposeEditor`.

Сохранение состояний редактора может быть реализовано путем объектов состояний (`FileEditorState`). При открытии файла IntelliJ IDEA пытается восстановить состояние редактора на момент закрытия файла путем вызова метода `readState`. При закрытии файла IntelliJ IDEA сигнализирует об этом вызовом метода `writeState`.

Пример реализации файлового редактора приведен в листинге 15.22.

Листинг 15.22

```
package org.intellij.samples.editor.fileEditor;

import com.intellij.openapi.fileEditor.FileEditor;
import com.intellij.openapi.fileEditor.FileEditorState;
```

```
import com.intellij.openapi.fileEditor.FileEditorStateLevel;

import javax.swing.*;
import java.beans.PropertyChangeListener;
import java.text.MessageFormat;

public class SampleFileEditor implements FileEditor {
    private String name;
    private JComponent component;

    public SampleFileEditor(String name, Icon icon) {
        this.name = MessageFormat.format(
            "{0} ({1}x{2})", new Object[]{
                name,
                new Integer(icon.getIconWidth()),
                new Integer(icon.getIconHeight())
            }
        );

        JLabel view = new JLabel(icon);
        view.setToolTipText(this.name);
        component = new JScrollPane(view);
    }

    public JComponent getComponent() {
        return component;
    }

    public JComponent getPreferredFocusedComponent() {
        return component;
    }

    public String getName() {
        return name;
    }

    public FileEditorState getState(FileEditorStateLevel level) {
```

```
        return new FileEditorState() {
        };
    }

    public void setState(FileEditorState state) {
    }

    public boolean isModified() {
        return false;
    }

    public boolean isValid() {
        return true;
    }

    public void selectNotify() {
    }

    public void deselectNotify() {
    }

    public void addPropertyChangeListener(
        PropertyChangeListener listener
        ) {
    }

    public void removePropertyChangeListener(
        PropertyChangeListener listener
        ) {
    }
}
```

При реализации редактора необходимо создать визуальные компоненты, которые и будут его составлять. При этом метод `getComponent` должен возвращать полностью весь визуальный объект, а метод `getPreferredFocusedComponent` может вернуть лишь рабочую область визуального компонента, к которой будет обращен фокус при открытии файла.

Метод `getName` задает наименование редактора, однако это имя будет показано, только если в системе зарегистрировано более одного редактора для одного и того же файла (рис. 15.8).

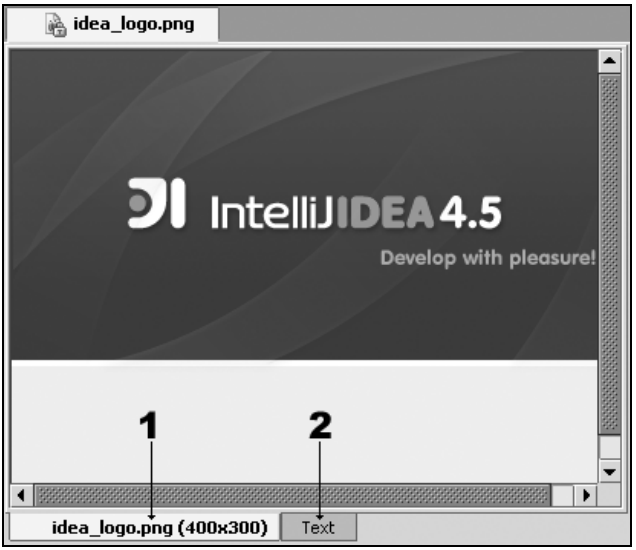


Рис. 15.8. Внешний вид редактора

На нижних закладках видны названия редакторов:

- закладка под номером **1** — реализация редактора приведена в листинге 15.22;
- закладка под номером **2** — стандартный текстовый редактор, в который загружено содержимое файла.

Редактор также может менять свои состояния в процессе работы. Метод `getState` должен возвращать состояние заданного уровня. Уровень передается в метод `getState` в виде объекта класса `FileEditorStateLevel`. Уровень может иметь два значения:

- `FULL` — полный снимок состояния редактора;
- `UNDO` — предыдущее состояние редактора.

Метод `setState` устанавливает заданное состояние редактора.

15.2.7.2. Использование редактора файлов

Использование стандартного редактора или, другими словами, открытие заданного файла в стандартном редакторе осуществляется с помощью класса `FileEditorManager` из пакета `com.intellij.openapi.fileEditor`. Пример использования данного класса представлен в листинге 15.23.

Листинг 15.23

```
package org.intellij.samples.editor.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.application.PathManager;
import com.intellij.openapi.fileChooser.FileChooser;
import com.intellij.openapi.fileChooser.FileChooserDescriptor;
import com.intellij.openapi.fileEditor.FileEditorManager;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.vfs.LocalFileSystem;
import com.intellij.openapi.vfs.VirtualFile;
import com.intellij.openapi.vfs.VirtualFileSystem;

import java.io.File;

public class SampleImageChooseAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        // Browse for images
        FileChooserDescriptor descriptor = new FileChooserDescriptor(
            true, false, false, false, true, false
        );
        VirtualFileSystem fileSystem = LocalFileSystem.getInstance();
        VirtualFile icons = fileSystem.findFileByPath(
            PathManager.getLibPath().replace(File.separatorChar, '/') +
            "/icons.jar"
        );
        descriptor.setRoot(icons);
        descriptor.setTitle("Choose image");
        VirtualFile[] virtualFiles = FileChooser.chooseFiles(
            project,
            descriptor
```

```
        );
        if (virtualFiles.length > 0) {
            VirtualFile virtualFile = virtualFiles[0];
            FileEditorManager manager = FileEditorManager.getInstance(
                project
            );
            manager.openFile(virtualFile, true);
        }
    }
}
```

Таким образом, с помощью класса `FileEditorManager` можно передать файл в редактор, который зарегистрирован в системе для данного типа файлов.

Для открытия файлов сразу во всех зарегистрированных редакторах можно использовать метод `openFile`, который принимает файл (`VirtualFile`) как параметр. Метод возвращает массив объектов класса `FileEditor` из пакета `com.intellij.openapi.fileEditor`. Массив на выходе данного метода обусловлен тем, что один файл может быть открыт несколькими редакторами сразу. Например, визуальные формы в IntelliJ IDEA открываются в редакторе форм, но также существует скрытый редактор, который отображает форму в виде XML.

Примечание

Если в параметрах запуска IntelliJ IDEA указать свойство `idea.is.internal=true`, то при редактировании форм можно видеть все скрытые редакторы, зарегистрированные для данного типа файлов.

Метод `openTextEditor` предназначен для открытия заведомо текстовых файлов. Метод возвращает объект класса `Editor` из пакета `com.intellij.openapi.editor`. Для открытия файла в редакторе необходимо указать дескриптор файла. Дескриптор файла определяет как сам файл, так и положение курсора в данном файле.

Примечание

Метод `openTextEditor` игнорирует свойство `HIDE_DEFAULT_EDITOR` и отдает предпочтение именно текстовому редактору, если для одного файла зарегистрировано более одного редактора.

15.2.7.3. Использование стандартного редактора текстовой информации

Для отображения текстовой информации и работы с ней в IntelliJ IDEA, в библиотеке OpenAPI, предусмотрен специальный класс, позволяющий открыть текстовый редактор для строки или для объекта класса `Document` из пакета `com.intellij.openapi.editor`.

Для создания редактора можно воспользоваться классом `EditorFactory` из пакета `com.intellij.openapi.editor` (листинг 15.24).

Листинг 15.24

```
package org.intellij.samples.editor.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.editor.Document;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.editor.EditorFactory;
import com.intellij.openapi.project.Project;

public class SampleTextEditorAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstant.PROJECT
        );
        EditorFactory factory = EditorFactory.getInstance();
        Document document = factory.createDocument("Hello World!");
        Editor editor = factory.createEditor(document, project);

        SampleTextEditorDialog dialog = new SampleTextEditorDialog(
            project, editor
        );
        dialog.pack();
        dialog.show();
        factory.releaseEditor(editor);
    }
}
```

Подсветка синтаксиса

Для подсветки синтаксиса в текстовом редакторе используются регион и уровни. Процесс подсветки текста достаточно прост — необходимо выделить регион, выбрать нужный уровень и атрибуты текста: оформление шрифта, различные эффекты и цветовые атрибуты.

Уровни необходимы для того, чтобы присвоение цвета шрифту происходило с перекрытием. Чем выше уровень, тем выше его приоритет. Присвоение цвета региону, выполненное на более низком уровне, будет игнорироваться, если на верхнем уровне был присвоен другой цвет. Аналогично происходит обычная покраска слоями, когда поверх одной краски наносится другая.

После разметки маркированные участки могут менять свои размеры, например, если символ вставляется внутрь маркированного региона, то вставляемый символ принимает его оформление, одновременно корректируя размеры вместе с границами маркировки. При удалении символов регион и маркированные границы уменьшаются на число удаленных символов.

Для маркировки регионов в редакторе используется класс `MarkupModel` из пакета `com.intellij.openapi.editor.markup` (листинг 15.25), который можно получить из текстового редактора методом `getMarkupModel`.

Листинг 15.25

```
package org.intellij.samples.editor.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.editor.Document;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.editor.EditorFactory;
import com.intellij.openapi.editor.markup.HighlighterLayer;
import com.intellij.openapi.editor.markup.HighlighterTargetArea;
import com.intellij.openapi.editor.markup.MarkupModel;
import com.intellij.openapi.editor.markup.TextAttributes;
import com.intellij.openapi.project.Project;

import java.awt.*;

public class SampleColoredTextEditorAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
```

```
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        EditorFactory factory = EditorFactory.getInstance();
        Document document = factory.createDocument("Hello Color World!");
        Editor editor = factory.createEditor(document, project);

        MarkupModel markupModel = editor.getMarkupModel();
        markupModel.addRangeHighlighter(
            0,
            5,
            HighlighterLayer.ADDITIONAL_SYNTAX,
            new TextAttributes(Color.BLUE, null, null, null, 0),
            HighlighterTargetArea.EXACT_RANGE
        );
        markupModel.addRangeHighlighter(
            6,
            11,
            HighlighterLayer.ADDITIONAL_SYNTAX,
            new TextAttributes(Color.GREEN, null, null, null, 0),
            HighlighterTargetArea.EXACT_RANGE
        );
        markupModel.addRangeHighlighter(
            12,
            17,
            HighlighterLayer.ADDITIONAL_SYNTAX,
            new TextAttributes(Color.RED, null, null, null, 0),
            HighlighterTargetArea.EXACT_RANGE
        );

        SampleTextEditorDialog dialog = new SampleTextEditorDialog(
            project, editor
        );
        dialog.pack();
        dialog.show();
        factory.releaseEditor(editor);
    }
}
```

Цветовые схемы

Цветовыми схемами в текстовых редакторах IntelliJ IDEA называются предопределенные стили оформления (`TextAttributes` из пакета `com.intellij.openapi.editor.markup` или `Color` из пакета `java.awt`). Каждому цвету или текстовому оформлению задается идентификатор.

Используя цветовые схемы в редакторах, можно позволить пользователю менять настройки подсветки синтаксиса, как это сделано в IntelliJ IDEA. При определении разметки в редакторе, цвета и оформления берутся из схемы по заданному идентификатору (листинг 15.26). Для стандартного редактора доступные идентификаторы определены в классе `EditorColors` из пакета `com.intellij.openapi.editor.colors`.

Листинг 15.26

```
package org.intellij.samples.editor.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.editor.Document;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.editor.EditorFactory;
import com.intellij.openapi.editor.colors.EditorColors;
import com.intellij.openapi.editor.colors.EditorColorsScheme;
import com.intellij.openapi.editor.markup.HighlighterLayer;
import com.intellij.openapi.editor.markup.HighlighterTargetArea;
import com.intellij.openapi.editor.markup.MarkupModel;
import com.intellij.openapi.project.Project;

public class SampleSchemeTextEditorAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        EditorFactory factory = EditorFactory.getInstance();
        Document document = factory.createDocument("Like search");
        Editor editor = factory.createEditor(document, project);
```

```
        EditorColorsScheme scheme = editor.getColorsScheme();
        MarkupModel markupModel = editor.getMarkupModel();
        markupModel.addRangeHighlighter(
            0,
            11,
            HighlighterLayer.ADDITIONAL_SYNTAX,
            scheme.getAttributes(
                EditorColors.TEXT_SEARCH_RESULT_ATTRIBUTES
            ),
            HighlighterTargetArea.EXACT_RANGE
        );

        SampleTextEditorDialog dialog = new SampleTextEditorDialog(
            project, editor
        );
        dialog.pack();
        dialog.show();
        factory.releaseEditor(editor);
    }
}
```

В библиотеке OpenAPI существует возможность получения цветовой схемы редактора и возможность внести изменения в полученную схему. Однако при использовании стандартной реализации цветовой схемы может не работать сохранение и восстановление схемы из XML-файлов. Для этого требуется создавать собственную имплементацию интерфейсов `EditorColorScheme` и `JDOMExternalizable`, в которой реализовывать необходимую логику сохранения и восстановления схемы.

Примечание

Библиотека PLUS имеет ряд вспомогательных классов для работы с цветовыми схемами, базовую реализацию цветовой схемы, которая может сохранять и восстанавливать свое состояние в формате цветовых схем IntelliJ IDEA. Кроме того, имеется компонента, которая позволяет визуально настраивать схемы для собственных редакторов. Функциональность этой библиотеки использована в плагине SystemTools для настройки цветов редактора-консоли. Библиотека доступна на сайте сообщества разработчиков по адресу **http://www.intellij.org/twiki/bin/view/Main/PlusLibrary.**

15.2.7.4. Система действий редактора

Система действий редактора аналогична системе действий IntelliJ IDEA. Различаются простые действия редактора и действия, которые могут изменять документ редактора. Изменения в редакторе возможны только в определенном потоке и поэтому должны запускаться с помощью метода `runWriteAction` класса `Application`.

Для удобства разработчиков не требуется реализовывать действия вручную, а достаточно лишь зарегистрировать их обработчики методом `setActionHandler` класса `EditorActionManager` из пакета `com.intellij.openapi.editor.actionSystem`. Идентификаторы действий редактора находятся в файле `IdeActions` из пакета `com.intellij.openapi.actionSystem` с префиксом `ACTION_EDITOR`.

Однако следует помнить, что регистрация собственного обработчика событий в редакторе заменит существующий стандартный обработчик, поэтому необходимо обеспечить делегирование необработанных событий стандартному обработчику.

Большинство данных действий носят характер перемещения по редактору, а также обработки специальных клавиш, таких как табуляция или <Enter> (листинг 15.27).

Однако существует еще одно специальное действие, которое предназначено для перехвата всех символьных событий клавиатуры. Данное действие представлено классом `TypedAction` и может быть получено методом `getTypedAction`. Для этого действия, как и для других, может быть задан собственный обработчик событий (*см. раздел 15.2.7.4*).

Листинг 15.27

```
package org.intellij.samples.editor.actionSystem;

import com.intellij.openapi.actionSystem.DataContext;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.editor.actionSystem.EditorActionHandler;
import com.intellij.openapi.editor.actionSystem.EditorWriteActionHandler;

public class SampleTabActionHandler extends EditorWriteActionHandler {
    private EditorActionHandler delegate;

    public SampleTabActionHandler(EditorActionHandler delegate) {
        this.delegate = delegate;
```

```
    }

    public void executeWriteAction(
        Editor editor, DataContext dataContext
        ) {
        SampleStatisticsManager manager =
            SampleStatisticsManager.getInstance();
        manager.incCharacter('\t');

        if (delegate != null) {
            delegate.execute(editor, dataContext);
        }
    }
}
```

Перехват символьных событий с клавиатуры

В IntelliJ IDEA существует специальный обработчик для событий ввода данных с клавиатуры, который перехватывает все события и направляет их в необходимый редактор.

Данный обработчик событий может быть заменен другим, с необходимым делегированием к стандартному обработчику. Иными словами, если требуется создать собственный обработчик, то стандартный ни в коем случае не должен игнорироваться, так как это может привести к блокировке ввода информации пользователем в стандартных редакторах.

Потребность в обработчике событий может возникнуть, когда необходимо фильтровать ввод пользователя, ограничить передвижения курсора внутри редактора или тем или иным образом наблюдать за вводом информации в редакторы IntelliJ IDEA.

С помощью метода `setupHandler` из класса `TypedAction` из пакета `com.intellij.openapi.editor.actionSystem` можно заменить стандартный обработчик на любую другую реализацию. Реализация обработчика состоит в том, чтобы обработать событие для заданного редактора (листинг 15.28).

Листинг 15.28

```
package org.intellij.samples.editor.actionSystem;

import com.intellij.openapi.actionSystem.DataContext;
import com.intellij.openapi.editor.Editor;
```

```
import com.intellij.openapi.editor.actionSystem.TypedActionHandler;

public class SampleTypedActionHandler implements TypedActionHandler {
    private TypedActionHandler delegate;

    public SampleTypedActionHandler(TypedActionHandler delegate) {
        this.delegate = delegate;
    }

    public void execute(
        Editor editor, char charTyped, DataContext dataContext
        ) {
        SampleStatisticsManager manager =
            SampleStatisticsManager.getInstance();
        manager.incCharacter(charTyped);

        if (delegate != null) {
            delegate.execute(editor, charTyped, dataContext);
        }
    }
}
```

15.2.8. Виртуальная файловая система IntelliJ IDEA (VFS)

В IntelliJ IDEA существует представление файловой системы, которая предназначена для реализации процедур ввода/вывода — представления файлов и директорий, которые присутствуют в проектных модулях. Такая система представления файлов называется виртуальной файловой системой (Virtual File System).

15.2.8.1. Общие сведения

В данный момент существует две реализации виртуальных файловых систем в IntelliJ IDEA:

- виртуальная файловая система, дублирующая файловую систему операционной среды;
- виртуальная файловая система, позволяющая работать с JAR- и ZIP-архивами как с обычными директориями.

Реализация файловой системы представляет собой реализацию абстрактного класса `VirtualFileSystem` из пакета `com.intellij.openapi.vfs`. В базовой реализации предусмотрено оповещение об изменениях в файловой системе посредствам вызова соответствующего метода в реализации (`fireFileCreated`, `fireFileDeleted`). Так, например, при создании файла необходимо оповестить VFS вызовом метода `fireFileCreated`, передав в качестве параметров объект, который вызвал создание файла (`requester`), и сам файл, только что созданный в системе. В любой реализации `VirtualFileSystem` должно присутствовать корректное оповещение о любом событии, иначе IntelliJ IDEA не узнает о том, что файл обновился.

Внимание

К написанию файловых систем стоит подходить осторожно, так как это основной код, отвечающий за сохранность данных на носителе. Ошибки в этом месте могут привести к потере данных. Также любое замедление и лишние вызовы метода `refresh` (этот метод должен обновлять всю видимую область виртуальной файловой системы) повлияют на общую производительность IntelliJ IDEA.

15.2.8.2. Использование VFS для работы с локальными файлами и архивами

В основном использование VFS направлено на поиск локальных файлов или файлов, расположенных внутри JAR- или ZIP-архивов, а также на работу с ними. В IntelliJ IDEA не используется файловая система напрямую. В вызовах всегда фигурирует класс `VirtualFile` из пакета `com.intellij.openapi.vfs`, а не сам `File` из пакета `java.io`. Поэтому при разработке плагинов необходимо работать с файлами через данную виртуальную файловую систему.

При поиске файлов следует учитывать тот факт, что разделителем пути файла является не системный разделитель, а символ `/`, поэтому при поиске файлов необходимо все разделители в пути файла преобразовать к символу `/`.

Поиск файла осуществляется функцией `findFileByPath` класса `VirtualFileSystem` (листинг 15.29).

Листинг 15.29

```
package org.intellij.samples.vfs.actions;

import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.DataConstants;
import com.intellij.openapi.application.PathManager;
```

```
import com.intellij.openapi.project.Project;
import com.intellij.openapi.ui.Messages;
import com.intellij.openapi.vfs.LocalFileSystem;
import com.intellij.openapi.vfs.VirtualFile;
import com.intellij.openapi.vfs.VirtualFileSystem;

import java.io.File;
import java.io.IOException;

public class SampleBuildNumberAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Project project = (Project)e.getDataContext().getData(
            DataConstants.PROJECT
        );
        VirtualFileSystem vfs = LocalFileSystem.getInstance();

        StringBuffer path = new StringBuffer(
            PathManager.getHomePath().replace(File.separatorChar, '/')
        );
        path.append("/build.txt");
        VirtualFile buildFile = vfs.findFileByPath(path.toString());
        if (buildFile != null) {
            try {
                Messages.showInfoMessage(
                    project,
                    "Build Number: " +
                    new String(buildFile.contentsToCharArray()),
                    "Build Info"
                );
            } catch (IOException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

При работе с виртуальными файлами можно использовать специальный утилитный класс `VfsUtil`, который содержит большое количество полезных

функций. Данный класс позволяет искать файлы по относительным путям, конвертировать пути в URL и обратно и т. п.

15.2.9. Система контроля версий IntelliJ IDEA (VCS)

Виртуальная система контроля версий это отдельная подсистема, позволяющая интегрировать системы контроля версий с IntelliJ IDEA. В пакете `com.intellij.openapi.vcs` находятся интерфейсы, которые следует имплементировать для интеграции IntelliJ IDEA с системой контроля версий. Данная подсистема носит сокращенное название VCS.

Для реализации VCS необходимо реализовать проектную компоненту, которая основывается на абстрактном классе `AbstractVcs` (листинг 15.30).

Листинг 15.30

```
package org.intellij.samples.vcs;

import com.intellij.openapi.components.ProjectComponent;
import com.intellij.openapi.options.Configurable;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.vcs.AbstractVcs;
import com.intellij.openapi.vcs.VcsException;

public class SampleVcs extends AbstractVcs implements ProjectComponent {
    private static final String NAME = "SampleVcs";

    public SampleVcs(Project project) {
        super(project);
    }

    public String getName() {
        return NAME;
    }

    public String getDisplayName() {
        return "Sample VCS";
    }

    public Configurable getConfigurable() {
```

```
        return new SampleVcsConfigurable();
    }

    public void checkinFile(String path, Object parameters)
        throws VcsException {
    }

    public void addFile(
        String folderPath, String name, Object parameters
        ) throws VcsException {
    }

    public void removeFile(String path, Object parameters)
        throws VcsException {
    }

    public void addDirectory(
        String parentPath, String name, Object parameters
        ) throws VcsException {
    }

    public void removeDirectory(String path, Object parameters)
        throws VcsException {
    }

    public void projectOpened() {
    }

    public void projectClosed() {
    }

    public String getComponentName() {
        return NAME;
    }

    public void initComponent() {
```

```
    }

    public void disposeComponent() {
    }
}
```

Чтобы наделить VCS дополнительными возможностями, по сравнению с базовой реализацией, необходимо имплементировать дополнительные интерфейсы:

- `DifferencesProvider` — поддержка просмотра и сравнения изменений;
- `DirectoryMoveProvider` — поддержка перемещения директорий;
- `DirectoryRenameProvider` — поддержка переименования директорий;
- `EditFileProvider` — поддержка функции редактирования файлов в системе;
- `FileMoveProvider` — поддержка перемещения файлов;
- `FileRenameProvider` — поддержка переименования файлов;
- `FileStatusProvider` — поддержка статуса файлов (`FileStatus`);
- `RollbackProvider` — поддержка откатов;
- `TransactionProvider` — поддержка транзакций;
- `UpToDateRevisionProvider` — поддержка запроса последней версии.

15.2.10. Структурный программный интерфейс IntelliJ IDEA (PSI)

IntelliJ IDEA имеет встроенную систему структуризации программного кода. Данная система позволяет работать с кодом в виде структурного иерархического дерева.

Для работы с данной системой используется структурный прикладной интерфейс (PSI — Program Structure Interface). Данный интерфейс представлен многочисленными классами в пакете `com.intellij.psi`, где каждый класс символизирует какой-либо элемент структурного дерева.

Существует специальное средство для просмотра и построения PSI-деревьев, которое облегчает разработку кода, связанного с PSI. Данное средство является частью закрытой функциональности IntelliJ IDEA и для его включения необходимо в параметрах запуска IntelliJ IDEA указать системное свойство `idea.is.internal=true`. После этого воспользуйтесь командой из меню **Tools | Internal Actions | View PSI Structure** (Утилиты | Внутренние действия | просмотр PSI-структуры) (рис. 15.9).

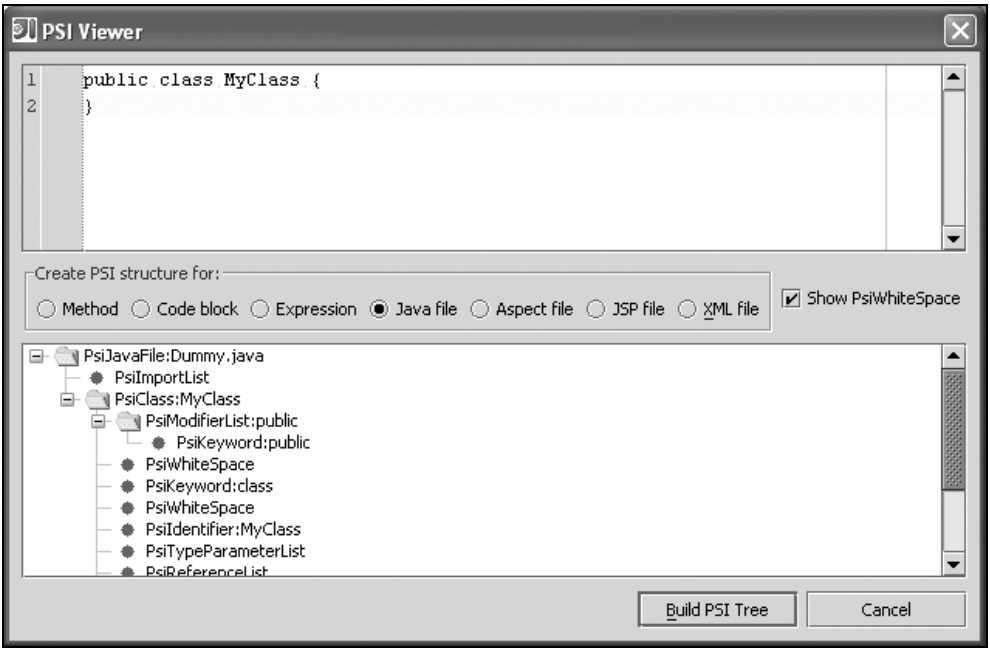


Рис. 15.9. Просмотр PSI-структуры

В данном примере показано, как код пустого класса будет представлен внутри IntelliJ IDEA. Таким IntelliJ IDEA видит любой код, для которого предусмотрена структура PSI. После преобразования текста в структурное дерево работа с кодом превращается в работу с деревом, что намного упрощает анализ и преобразования кода.

15.2.10.1. Инспекции кода (Inspections)

Инспекции кода в редакторах необходимы для выявления проблем "на лету". Инспекции кода основываются на PSI-структуре и предоставляют разработчику возможность определить самому участки PSI-структуры, которые должны быть подсвечены.

Для реализации инспекций необходимо имплементировать интерфейс `InspectionToolProvider` из пакета `com.intellij.codeInspection`, который должен быть компонентой приложения. Имплементация `InspectionToolsProvider` определяет, какие инспекции (реализации абстрактного класса `LocalInspectionTool`) доступны.

Примечание

В плагине Inspection Gadgets содержится множество примеров инспекций и преобразований. Исходные коды данного плагина включены в состав Plugin Development Package IntelliJ IDEA.

15.2.10.2. Преобразования кода (Intentions)

Преобразования кода ("подсказки") доступны в стандартных редакторах IntelliJ IDEA и предназначены для локального преобразования кода. Для регистрации новых преобразователей необходимо имплементировать интерфейс `IntentionAction` и передать реализацию в метод `addAction` класса `IntentionManager` из пакета `com.intellij.codeInsight.intention`.

Преобразования должны основываться на PSI-структуре. Однако простые преобразования можно делать и без PSI (листинг 15.31).

Листинг 15.31

```
package org.intellij.samples.psi.intentions;

import com.intellij.codeInsight.intention.IntentionAction;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.editor.SelectionModel;
import com.intellij.openapi.project.Project;
import com.intellij.psi.PsiFile;
import com.intellij.util.IncorrectOperationException;

public class SampleInverseSelectionIntention implements IntentionAction {
    public String getText() {
        return "Inverse symbols";
    }

    public String getFamilyName() {
        return "Sample";
    }

    public boolean isAvailable(
        Project project, Editor editor, PsiFile file
        ) {
        SelectionModel selectionModel = editor.getSelectionModel();
        return !selectionModel.hasBlockSelection() &&
            selectionModel.hasSelection();
    }

    public void invoke(Project project, Editor editor, PsiFile file)
```

```
        throws IncorrectOperationException {
        SelectionModel selectionModel = editor.getSelectionModel();
        int start = selectionModel.getSelectionStart();
        int end = selectionModel.getSelectionEnd();
        String text = selectionModel.getSelectedText();
        char[] chars = text.toCharArray();
        for (int i = 0; i < chars.length >> 1; i++) {
            char aChar = chars[i];
            chars[i] = chars[chars.length - i - 1];
            chars[chars.length - i - 1] = aChar;
        }
        editor.getDocument().replaceString(start, end,
            new String(chars));
    }

    public boolean startInWriteAction() {
        return true;
    }
}
```

Примечание

В плагине Intention Power Pack содержится множество примеров действий преобразования. Исходные коды данного плагина включены в состав Plugin Development Package IntelliJ IDEA.

15.3. Структура директорий плагина

Плагин может быть представлен как одним JAR-архивом, так и упакованной в ZIP-архив директорией с несколькими JAR-архивами. Путь к общей директории плагинов задается свойством при запуске IntelliJ IDEA — `idea.plugins.path`. По умолчанию данное свойство указывает на директорию plugins внутри конфигурационной директории IntelliJ IDEA.

Если плагин представлен только одним JAR-файлом, то он располагается непосредственно в директории плагинов:

```
<idea.plugins.path>
  plugin01.jar
  plugin02.jar
  plugin03.jar
```

В общем случае, каждый плагин имеет отдельную директорию. В данной директории IntelliJ IDEA автоматически загрузит все классы и ресурсы из директории classes и из каждого JAR-архива внутрь поддиректории lib:

```
<idea.plugins.path>
  plugin01
    classes
      MyClass.class
    lib
      plugin.jar
      lib.jar
  plugin02
    classes
      MyClass.class
    lib
      plugin.jar
      lib.jar
```

15.4. Дескриптор плагина

Дескриптор плагина (plugin.xml) используется для описания компонент плагина, информации о плагине, авторе и действиях плагина. Дескриптор плагина представляет собой XML-файл, который располагается в директории META-INF в любом из JAR-архивов в дистрибутиве плагина или в директории плагина.

15.4.1. Требования к дескрипторам

Дескриптор должен удовлетворять DTD для дескриптора плагина. Для этого необходимо включать в начало XML-файла ссылку на DTD-файл (листинг 15.32).

Листинг 15.32

```
<?xml version='1.0'?>
<!DOCTYPE idea-plugin SYSTEM "http://plugins.intellij.net/plugin.dtd">
```

15.4.2. Общая информация о плагине

Описание плагина начинается с элемента `idea-plugin`. В общей секции указывается информация о плагине, информация об авторе и о поддерживаемых версиях.

15.4.2.1. Элемент *idea-plugin*

`idea-plugin` — это корневой элемент в дескрипторе плагинов. Дополнительно можно указать атрибут `url`, содержащий URL домашней страницы плагина, как правило, это страница сайта сообщества разработчиков плагинов IntelliJ IDEA.

15.4.2.2. Элемент *name*

Элемент `name` вложен в корневой элемент дескриптора плагина и определяет имя плагина.

15.4.2.3. Элемент *version*

Элемент `version` вложен в корневой элемент дескриптора плагина и определяет версию плагина.

15.4.2.4. Элемент *description*

Элемент `description` вложен в корневой элемент дескриптора плагина и содержит краткое описание плагина.

15.4.2.5. Элемент *change-notes*

Элемент `change-notes` вложен в корневой элемент дескриптора плагина и содержит историю изменения плагина.

15.4.2.6. Элемент *vendor*

Элемент `vendor` вложен в корневой элемент дескриптора плагина и содержит информацию о разработчике плагина. Атрибуты:

- `email` — определяет адрес электронной почты разработчика;
- `url` — определяет URL домашней страницы разработчика. Как правило, это адрес пользовательской страницы на сайте сообщества разработчиков плагинов IntelliJ IDEA.

15.4.2.7. Элемент *idea-version*

Данный элемент вложен в корневой элемент дескриптора плагина и содержит информацию о поддерживаемой версии IntelliJ IDEA.

Атрибуты:

- `since-build` — номер сборки IntelliJ IDEA, с которым данный плагин будет работать корректно. Обычно, это номер сборки, на котором производится разработка плагина;
- `until-build` — номер сборки IntelliJ IDEA, до которого плагин работает на всех IntelliJ IDEA. Здесь указывается номер сборки, начиная с которого плагин может работать некорректно.

15.4.3. Описания компонент

Компоненты плагина описываются в соответствии с реализацией:

- `application-components` — описание всех компонент уровня приложения;
- `project-components` — описание всех компонент проектного уровня;
- `module-components` — описание всех модульных компонент.

Содержание данных элементов идентично. Каждая компонента описывается вложенным элементом `component` в соответствии с требуемым уровнем (листинг 15.33).

Листинг 15.33

```
<application-components>
    <component>
        <implementation-class>org.MyClass</implementation-class>
    </component>
</application-components>
```

Внимание

При указании имен классов нельзя делать переводы строк, так как загрузчик плагинов будет искать класс с переводами строк в его имени и, скорей всего, не найдет его.

15.4.3.1. Элемент *interface-class*

Данный элемент вложен в элемент `component` дескриптора плагина и содержит информацию об интерфейсе, который имплементирует компоненту. Если указан элемент `interface-class`, то в метод `getComponent` для поиска данной компоненты необходимо передавать именно указанный интерфейс.

15.4.3.2. Элемент *implementation-class*

Данный элемент вложен в элемент `component` дескриптора плагина и содержит информацию о реализации компонента.

15.4.4. Описание действий

Для автоматической регистрации действий плагина в дескрипторе предусмотрены соответствующие элементы. При описании действий плагина можно сразу включать действия в стандартные меню и панели инструментов. Данное расширение дескриптора плагина намного упрощает разработку действий.

Действия плагина описываются внутри элемента `actions` (листинг 15.34).

Листинг 15.34

```
<actions>
    <group text="Samples">
        <action
             class="org.MyAction"
             id="MyAction"
             text="My Action"
             icon="/myActions/icon.png" />
        <add-to-group
            relative-to-action="HelpMenu"
            anchor="after"
            group-id="MainMenu" />
    </group>
</actions>
```

15.4.4.1. Элемент *group*

Данный элемент объединяет действия в группу. Атрибуты:

- `id` — идентификатор группы для организации ссылок;
- `text` — наименование группы. Если это меню, то данный атрибут определяет наименование меню;
- `icon` — пиктограмма для группы;
- `popup` — атрибут, определяющий, является ли данная группа действий всплывающим меню или нет. Возможные значения `true` и `false`.

15.4.4.2. Элемент *action*

Данный элемент описывает действие. Атрибуты:

- `class` — класс имплементации действия;
- `id` — идентификатор действия для организации ссылок;
- `text` — наименование действия;
- `description` — описание действия;
- `icon` — пиктограмма для действия.

15.4.4.3. Элемент *reference*

Данный элемент описывает ссылку на действие или группу по идентификатору. Атрибуты:

- `id` — идентификатор действия или группы.

15.4.4.4. Элемент *add-to-group*

Данный элемент определяет контейнер для своего родительского элемента. Атрибуты:

- `group-id` — идентификатор группы, куда необходимо добавить родительский элемент (действие или группу действий);
- `relative-to-action` — задает положение относительно указанного действия;
- `anchor` — тип расположения относительно действия, указанного в атрибуте `relative-to-action`. Возможные значения:
  - `after` (после указанного действия);
  - `before` (перед указанным действием);
  - `first` (в начале группы);
  - `last` (в конце группы).

  Для значений `first` и `last` атрибут `relative-to-action` не обязателен.

15.4.4.5. Элемент *separator*

Данный элемент определяет разделитель в группе действий

15.4.4.6. Элементы *keyboard-shortcut* и *mouse-shortcut*

Данные элементы определяют комбинации клавиш для заданного действия (листинг 15.35).

Листинг 15.35

```
<action
    class="org.MyAction"
    id="MyAction"
    text="My Action"
    icon="/myActions/icon.png">
    <keyboard-shortcut
        first-keystroke="control alt M"
        second-keystroke="Y"
        keymap="$default" />
</action>
```

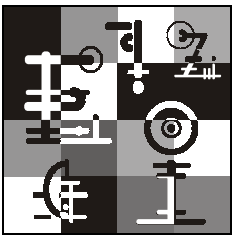
15.5. Заключение

Разработка плагинов позволяет узнать об IntelliJ IDEA гораздо больше, чем обычная разработка. Сама библиотека OpenAPI содержит огромное количество шаблонов программирования, и разрабатывая плагины для IntelliJ IDEA, можно найти очень много интересных подходов и нестандартных решений.

Разработчики компании JetBrains, которые на протяжении четырех с лишним лет продолжают добавлять к IntelliJ IDEA все новые и новые возможности, постоянно обновляют состав OpenAPI. В составе библиотеки появляются новые интересные модули. При этом, к сожалению, иногда утрачивается поддержка старых модулей. На этот случай можно использовать механизм поддержки номеров сборки IntelliJ IDEA в дескрипторе плагина (*см. раздел 15.4.2.7*).

Все примеры, приведенные в данной главе, вы можете найти на прилагаемом компакт-диске. На сайте сообщества разработчиков вы также можете найти огромное количество исходных текстов самых разнообразных плагинов.

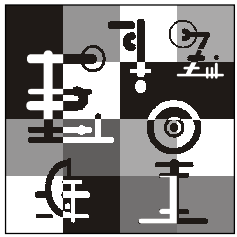
Удачи в разработке плагинов!



Часть IV

Ближайшее будущее

Глава 16. IDEA 5



Глава 16

IDEA 5

Цикл разработки новой версии IntelliJ IDEA занимает в среднем от полугода до года. Версия 4.5 была выпущена в июле 2004 года, спустя почти полгода с момента выпуска версии 4.0.

Следующая версия IntelliJ IDEA 5.0, скорее всего, будет выпущена весной или в начале лета 2005 года. Несмотря на то, что IntelliJ IDEA 5.0 еще находится в стадии разработки, уже сейчас можно попробовать новые возможности этой версии с помощью программы предварительного доступа EAP (*см. раздел 2.1*). На момент написания этой главы в предварительном доступе находится IntelliJ IDEA 5.0 сборка номер 3117.

В этой главе мы хотим рассказать о тех новых функциях и возможностях, которые появятся в IntelliJ IDEA версии 5.0.

Последнюю информацию о версии 5.0 можно получить на сайте IntelliJ Technology Network **http://www.intellij.net**.

16.1. Поддержка XML, HTML, CSS

В новой версии IntelliJ IDEA значительно улучшена поддержка форматов XML, HTML (в том числе и XHTML) и CSS, что должно порадовать разработчиков, использующих IntelliJ IDEA для Web-разработки.

16.1.1. XML

В XML улучшена поддержка типизации. Теперь при завершении XML-кода IntelliJ IDEA будет автоматически вставлять обязательные вложенные теги.

Будет доступна генерация XMLSchema под DTD, а также на основе произвольного XML-файла. Появится и обратная функция — генерация шаблона XML-файла по DTD или XMLSchema.

В XML теперь будут доступны некоторые рефакторинги (какие именно, пока не известно, но, вероятно, это будут команды перемещения и копирования, как минимум).

При редактировании XML-документа, для которого задана схема, можно будет с помощью стандартной команды получения комментария (<Ctrl>+<Q>) (*см. раздел 3.2.4.7*) вывести информацию о текущем элементе, если в схеме для этого элемента задана аннотация.

16.1.2. HTML

Для HTML теперь поддерживается команда простого завершения кода (*см. раздел 3.2.6.5*) для имен тегов и их атрибутов. Также будет доступна команда переформатирования (*см. раздел 3.2.11.1*), форматирующая HTML-файл в соответствии со стилями кода (*см. раздел 4.1.7*), которые будут доступны и для HTML.

Ко всему прочему появится возможность выполнять проверку HTML-файлов (validation).

16.1.3. CSS

CSS (Cascading Style Sheets — таблица каскадных стилей) будет полностью поддержана на уровне редактора в IntelliJ IDEA. Для CSS станут доступны команды автоматического завершения кода, получения информации о компонентах CSS, поиск обращений (*см. раздел 3.2.3.7*), а также подсветка неиспользованных классов CSS.

16.2. Поддержка J2EE

Некоторые новые возможности появятся и в поддержке J2EE. Так, будет возможность соотнести файлы с комментариями JavaDoc и файлы с исходными текстами для библиотек сервера приложений (в версии 4.5 это можно сделать только для обычных библиотек). Для J2EE-приложения появится возможность задать EJB-connector (адаптер ресурсов).

Также планируется реализовать поддержку ссылок в JSF (Java Server Faces) на уровне команд автоматического завершения, поиска обращений, рефакторингов и подсветки.

Одна из самых ожидаемых новых возможностей IntelliJ IDEA 5.0 — улучшенная поддержка JSP (в том числе JSP 2.0). Для JSP будут доступны новые команды рефакторинга (как для самого JSP, так и для фрагментов Java-кода). Ожидается поддержка JSTL (Java Standard Template Library), возможности оптимизации импортов, просмотр структуры JSP-файла, команды форматирования.

Насколько мы знаем, поддержка JSP переделывается достаточно основательно.

16.3. Работа с проектом

Улучшения и изменения коснутся и работы с проектом в IntelliJ IDEA. Так, планируется добавить дополнительную возможность создания списка избранных ссылок (аналог списка "Избранное" в браузере), в который можно добавлять ссылки на произвольные элементы проекта (сам проект, модули, директории, файлы, J2EE-компоненты и пр.). Называться этот список будет **Shelf** (Полка).

Для окна структуры проекта будет доступна возможность просматривать имена пакетов в сокращенном виде.

Одним из самых заметных и, на наш взгляд, удобных улучшений в организации проекта будет возможность организовывать модули проекта в группы, что позволит визуально значительно сократить список модулей, число которых для больших проектов может достигать нескольких десятков.

Также появится возможность импортировать проекты из IDE Eclipse — одного из самых сильных конкурентов IntelliJ IDEA.

Для проектов станет доступна поддержка циклических зависимостей между модулями, в том числе и при генерации скрипта сборки Ant (*см. раздел 3.2.9.5*).

16.4. Настройки

К числу дополнительных настроек добавится возможность настраивать "подсказки" (*см. раздел 6.1.3*).

Появится возможность настраивать панель инструментов главного окна IntelliJ IDEA (*см. раздел 3.3*).

При работе под операционной системой Mac OS X появится возможность выбирать JRE, под которым будет работать IntelliJ IDEA.

16.5. Рефакторинги

И без того большой список доступных рефакторингов пополнится дополнительными командами, некоторые из которых будут расширять функции существующих команд, другие будут совершенно новыми, например, преобразование группы констант в перечислимый тип (`enum`).

16.6. Системы контроля версий

По многочисленным запросам пользователей в новой версии IntelliJ IDEA появится поддержка двух новых систем контроля версий:

- Perforce;
- Subversion.

Данные системы контроля версий для текущей версии поддерживаются на основе плагинов (*см. главу 15*). Однако включение их поддержки в саму IntelliJ IDEA позволит значительно улучшить работу с ними.

К слову сказать, теперь будет возможно сконфигурировать ту или иную систему контроля версий не только для всего проекта, но и конкретно для каждого модуля.

В имеющейся поддержке системы контроля версий CVS будут добавлены новые функции для объединения ветвей (merge branches), а также возможность создавать и применять заплатки (patches).

16.7. Плагины и OpenAPI

Много изменений и улучшений затронет поддержку плагинов (*см. главу 15*).

Теперь в IntelliJ IDEA появится встроенная система создания и отладки плагинов. Во-первых, появится отдельный тип модуля — плагин, который будет включать в себя все необходимые функции для разработки плагина, подключение библиотек, запуск и отладку плагина без необходимости перезапуска IntelliJ IDEA.

Значительно будет расширен открытый интерфейс для разработки плагинов (OpenAPI), к которому добавятся такие составляющие:

- доступ к браузеру;
- просмотр структуры (*см. раздел 3.4.8.8*);
- типы файлов (*см. раздел 4.1.11*);
- функции "живых шаблонов", определенные пользователем (*см. раздел 4.1.13*);
- доступ к статистике продуктивности (*см. раздел 3.2.13.3*);
- расширение режима просмотра переменных при отладке;
- экспорт интерфейсов между плагинами;
- инспекция не-Java-файлов;
- интерфейс к скремблятору (Scramble API).

16.8. Поддержка J2ME

Также в новой версии IntelliJ IDEA появится долгожданная поддержка J2ME-приложений. Поддержка будет осуществляться непосредственно на уровне проекта и его модулей. Для мобильных приложений будет доступна отладка и запуск. Поддерживаемые платформы:

- MIDP 1.0/2.0;
- NTT DoCoMo i-mode.

16.9. Отладка

К дополнительным возможностям отладки приложений добавится новый тип точки останова — logging breakpoint — точка останова, выводящая в консоль некоторую строку.

Можно будет перетаскивать точки останова (drag and drop) в пределах одного файла. Функция перетаскивания будет также доступна при вычислении выражений.

Станет возможным сконфигурировать отдельный JDK персонально для каждой конфигурации запуска и отладки.

16.10. Дополнительные функции редактора

И без того отличный редактор IntelliJ IDEA также получит дополнительные функции. Например, такие, как возможность настраивать цвета для маркеров ошибок и замечаний, новую систему разделения закладок в редакторе, а также многочисленные косметические изменения.

Будут улучшены возможности навигации и перехода к классу, файлу и символу (*см. раздел 3.2.5.1—3.2.5.3*). Теперь будет производиться автоматический поиск в библиотеках в случае, если в проекте ничего не было найдено.

Дополнительно будет улучшена система автоматического завершения кода. "Живые шаблоны" смогут "запоминать" пользовательские предпочтения в именовании переменных. Простое завершение кода (*см. раздел 6.1.1.1*) будет работать и для обычного текста в строковых литералах, комментариях и пр.

Некоторые изменения коснутся стилей и форматирования кода.

16.11. Прочие улучшения

Кроме перечисленных выше улучшений и изменений, будет исправлено значительное количество ошибок старой версии (несомненно, будут внесены и новые ошибки, но, мы уверены, команда разработчиков IntelliJ IDEA должна быстро с ними справится ☺).

Заключение

С каждым днем скорость создания программного обеспечения растет. Новые возможности, функции и интерфейсы появляются настолько быстро, что ни успеваешь не только их изучить, но даже узнать о том, что они есть.

IntelliJ IDEA в этом плане не является исключением. Вместе с выходом новых версий IntelliJ IDEA, информация, представленная в этой книге, несомненно будет устаревать.

Однако не вся информация устареет. Может измениться интерфейс, появиться множество новых функций, но дух IntelliJ IDEA (не побоимся этого слова), как продукта, созданного Java-разработчиками для Java-разработчиков, останется неизменным. Неизменным останется и то удовольствие, которое испытывает каждый Java-разработчик от работы с IntelliJ IDEA.

Будем же разрабатывать с удовольствием ☺.

("Develop with pleasure")

Список ссылок

В данном разделе мы решили собрать все ссылки на сайты, которые приведены в книге. Можно рассматривать его как своеобразный "список использованной литературы" — к другим источникам при создании данной книги мы не обращались.

http://www.jetbrains.com — официальный сайт компании JetBrains, Inc., создателя IntelliJ IDEA.

http://www.intellij.net — IntelliJ Technology Network, сайт компании JetBrains, Inc., на котором публикуются предварительные версии IntelliJ IDEA, а также содержится система учета ошибок (bug-tracking).

http://www.intellij.org — сайт сообщества разработчиков IntelliJ IDEA.

http://www.livejournal.com/community/ideabook — сообщество в "Живом Журнале", посвященное нашей книге об IntelliJ IDEA.

http://java.sun.com — официальный сайт компании Sun Microsystems Inc., посвященный Java.

http://www.refactoring.com — сайт Мартина Фаулера (Martin Fowler), посвященный рефакторингу.

http://www.ibm.com/developerworks/oss/jikes — сайт компилятора Jikes фирмы IBM.

http://jakarta.apache.org — сайт проекта Apache/Jakarta. На этом сайте представлены проекты Ant и Tomcat, интеграция с которыми есть в IntelliJ IDEA, а также многие другие.

http://jakarta.apache.org/velocity/user-guide.html — описание языка шаблонов Velocity проекта Apache/Jakarta. Данные шаблоны используются в шаблонах файлов IntelliJ IDEA.

http://www.bea.com — сайт компании Bea, Inc. — производителя сервера приложений WebLogic.

http://www.junit.org — сайт проекта unit-тестирования Java классов JUnit.

http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html — информация об алгоритме вычисления цикломатической сложности. Этот алгоритм используется в одной из инспекций.

http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html — информация о "двойной блокировке", проверкой которой занимается одна из инспекций IntelliJ IDEA.

http://www.sourceforge.net — огромный хостинг Open Source-проектов.

Содержимое компакт-диска

К книге прилагается компакт-диск, на котором представлены все файлы проектов IntelliJ IDEA, а также исходные тексты всех классов, которые мы использовали при описании функциональности IntelliJ IDEA в главах этой книги. Все файлы проектов расположены внутри директории \Projects.

Также на компакт-диске представлен дистрибутив IntelliJ IDEA версии 4.5.3, любезно предоставленный нам для публикации. Файлы дистрибутива расположены в директории \IDEA.

В качестве примеров проектов для IntelliJ IDEA мы поместили на диск две версии JDK (1.4.2_04 и 1.5) для операционных систем Windows и Linux.

Таблица 1. *Содержимое CD*

Директория	Описание	Главы
\cvsTestProject	Проект, использующий интеграцию с системой контроля версий CVS (файлы проекта IntelliJ IDEA, а также файлы Java-классов)	12
\ssTestProject	Проект, использующий интеграцию с системой контроля версий SourceSafe (файлы проекта IntelliJ IDEA, а также файлы Java-классов)	12
\JDKDemo	Проект, использующий в качестве директорий исходных файлов примеры Java-классов из JDK (файлы проекта)	5
\IdeaBook	Проект, содержащий все файлы и Java-классы, использованные в качестве примера демонстрации возможностей IntelliJ IDEA в этой книге (файлы проекта, модулей, Java-классы)	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
\IdeaBook\ch15	Проекты с примерами плагинов IntelliJ IDEA	15
\IDEA	Дистрибутивы IntelliJ IDEA 4.5.3 для операционных систем Windows, Linux, Mac OS X и Unix	2
\JDK	Дистрибутивы Sun JDK 1.4.2_04 и 1.5.0 для операционных систем Windows и Linux	

Предметный указатель

A

AbstractVcs 766
ActionManager 727
  createActionPopupMenu 727
  createActionToolbar 728
ActionPopupMenu 727
  getComponent 727
ActionToolbar 728
  getComponent 717, 728
AnAction 723
  actionPerformed 723, 725
AnActionEvent
  getDataContext 723
  getPresentation 724
Annotation 65, 610, 618
  deprecated 65, 177
  override 75, 337
Ant 90, 232, 341, 436, 677, 699, 708, 783
  build.properties 701, 702
  генерация 690
  запуск 689
  привязка к проекту 687
  цель deploy 701
  цель native2ascii 708
  цель release 701
Applet 92, 517, 540
Appletviewer 520, 542
Application Server 93, 250, 299, 305, 526, 531, 533, 583
ApplicationComponent 710
Apt 618
Autoboxing 372, 610, 613
Autounboxing 372, 385, 610, 613
AWT 398

B

BaseComponent 710
  disposeComponent 710—712, 714
  getComponentName 710
  initComponent 710, 712, 714, 717
BEA 253
Bookmark. *См.* Закладка
Borland JBuilder 398, 413
Build.xml 677
Busy-waiting 382

C

CamelHumps 162
Class:
  enumeration 45, 223
  singleton 47, 223
  компиляция 512
  перейти к классу 71
  создать 44
Clipboard 343
Completion 83. *См.* Завершение кода
ComponentManager 709
  getComponent 774
ComponentWithBrowseButton 746
Configurable 731
  apply 733
  isModified 733
  reset 733
CSS 782
CVS 36, 176, 232, 249, 623, 631, 784
  add 657
  annotate 661
  branch 662
  check out 36, 655
  check status 659
  commit 650
  Edit and Watch 663
  history 659, 660
  import 36
  merge 656
  rollback local changes 655
  tag 662
  update 655

глобальные настройки 634, 654
импорт проекта 642
репозиторий 36, 635, 646
сравнение 658
Cyclomatic 356, 367

D

Deadlock 382
DefaultActionGroup 728
DefaultJDOMExternalizer 719, 721, 722
Deployment 227, 275, 288, 300, 305, 527
DialogWrapper 735
createActions 736, 737
createCenterPanel 735, 737
createContentPane 736, 737
createJButtonForAction 736, 737
createLeftSideActions 736
createNorthPanel 736, 737
createSouthPanel 736, 737
createTitlePanel 736, 737
getDimensionServiceKey 737
Diff. *См.* Файл: сравнение
DimensionService 710, 737
Document 756
DTD 106, 580—582, 781

E

EAP 25, 101
Eclipse 783
Editor 755
getMarkupModel 757
EditorActionManager 761
getTypedAction 761
setActionHandler 761
EditorColors 759
EditorColorScheme 760
EditorFactory 756
EJB 583
связи 585, 598
создание 585
Emacs 147, 229
Enum 48, 373, 610—614, 783

F

FAR 311
FileChooser 744
chooseFiles 744
FileChooserDescriptor 745
FileChooserDescriptorFactory 745
FileEditor 750—755
getComponent 752
getName 753
getPreferredFocusedComponent 752
getState 753
setState 753
FileEditorManager 753—755
openFile 755
openTextEditor 755
FileEditorPolicy
HIDE_DEFAULT_EDITOR 755
FileEditorProvider 747—750
accept 747
createEditor 747
readState 750
writeState 750
FileEditorState 750
FileEditorStateLevel 753
FULL 753
UNDO 753
Folding 70, 164, 176, 343
Forms Runtime 704
forms_rt.jar 708

G

Generics 89, 101, 215, 218, 262, 396, 508, 610, 611
GUI Designer 203, 232, 266, 427, 708, 733
Мастер связей 400, 434

H

Hot keys 32
HotSwap 21, 93, 579
HTML 140, 178, 542, 747, 781, 782
экспорт 36

I

IDE 21, 155, 398, 783
IDL 203
IIOP 263
Indent 85, 182
Inspection Gadgets 108, 171, 249, 769
InspectionToolProvider 769
Intention Action. *См.* Подсказка

Intention Power Pack 250, 255, 336, 770
IntentionAction 770
IntentionManager 770
Interface:
  annotation 48
  создать 44
Internationalization 365, 437
ITN 26, 781

J

J2EE 194, 226, 281, 580, 782, 254, 284
  Entity Bean 195, 226, 586, 591
  Filter 195, 602, 605
  Message Bean 195, 227, 597
  Servlet 23, 195, 227, 272, 602—604
  Session Bean 195, 227, 596
  WEB-INF 275
  приложения 600, 608
J2ME 785
  MIDP 785
Java 5, 48, 89, 101, 216, 220, 337, 373, 384, 385, 508, 610, 614
JavaDoc 64, 67, 71, 76, 99, 127, 158, 163—169, 177, 192, 290, 342, 349, 444, 682
JDK 26, 101, 195, 512, 524, 684, 699
  выбор 270
  модуль 293
  проект 293
JDOMExternalizable 710—721, 760
  InvalidDataException 719
  other.xml 722
  readExternal 710—714, 719
  writeExternal 711—714, 718
  WriteExternalException 718
JetBrains 15, 21—25, 113, 247
JPDA 549
JPopupMenu 727
JRE 26, 27
JSF 782
JSP 23, 163, 178, 182, 227, 272, 301, 531, 600, 606, 747
  JSTL 782
  TagLib 606
  валидация 514
  версия 2, 782
JSR 14, 45, 93, 101, 250, 297, 510, 515, 530, 583, 608—610
JUnit 93, 257, 362, 366, 515, 520, 544, 692
  TestCase 692
  запуск тестов 693

L

Linux 36
  установка 26
Live Template. *См.* Живой шаблон
Local History 106, 156, 623
LocalInspectionTool 769
Logger 715, 716
  error 717
  idea.log 717
Look&Feel 68, 159, 733

M

Mac OS X 159, 229, 783
  установка 27
MarkupModel 757
MDI 30
Messages 738
META-INF 285, 772
Microsoft Visual Studio 229, 311
Module 714
ModuleComponent 713
  moduleAdded 714

N

NamedJDOMExternalizable 718, 722
native2ascii 440, 708

O

OpenAPI 699, 704, 715, 728, 731, 747, 760, 784

P

Package:
  создать 45
Perforce 784
PHP 311
Plugin Development Package.
  См. OpenAPI

plugin.xml:
action 775
actions 775
add-to-group 776
application-components 774
change-notes 773
component 774
description 773
group 775
idea-plugin 773
idea-version 773
inplementation-class 774
interface-class 774
keyboard-shortcut 776
module-components 774
mouse-shortcut 776
name 773
project-components 774
reference 776
separator 776
vendor 773
version 773
Productivity Guide 112, 321
Program Structure Interface. *См.* PSI
ProjectComponent 711
project.default.xml 723
projectClosed 712, 714
projectOpened 712, 714
PropertiesComponent 710
PSI 768, 769
просмотр структуры 768

Q

Quick Fix 140, 318, 337
Quick Lists 233

R

Resource Bundle 403, 437
RMI 263, 514

S

Sandbox. *См.* Плагин: песочница
serialver 240
Shelf 783
SourceSafe 232, 623, 665
add 670
check in 670
check out 669
Explorer 669
Visual 665
сравнить 672
Splitter 746
SSH 639
StarTeam 232, 250, 623, 672
add 674
check in 675
check out 674
history 675
lock 675
unlock 675
сравнить 676
StatusBar. *См.* Панель статуса
Subversion 784
Swing 21, 104, 398, 416, 727, 728, 735

T

TCP/IP 524
TextAttributes 759
TextFieldWithBrowseButton 746
ToggleAction 723
isSelected 725
setSelected 725
Tomcat 93, 250, 515, 526, 546, 583
ToolBar. *См.* Панель инструментов
ToolWindow 739
ToolWindowAnchor 739
BOTTOM 739
LEFT 739
RIGHT 739
TOP 739
ToolWindowId 739
ToolWindowManager 739
registerToolWindow 739, 743
Transport:
Shared memory 524
Socket 524
Trove4J 704, 746
TypedAction 761, 762
setupHandler 762

U

Unix 182, 239
установка 27

V

Varargs 610, 616
VfsUtil 765
Vi 148, 311
VirtualFile 755, 764
VirtualFileSystem 764
  findFileByPath 764
  fireFileCreated 764
  fireFileDeleted 764
Visual J++ 398
Visual Slick Edit 311

W

WebLogic 93, 227, 250, 253, 278, 285, 288, 515, 533, 583, 591
Wiki 698
Windows 36, 159, 182, 200—205, 239
  установка 26
Wrapping 185

X

XML 28, 64, 85, 91, 106, 169, 178, 182, 202, 269, 272, 341, 398, 580, 678, 747, 781
  Schema 202, 580, 582, 781
  проверка 581
  редактирование 580

Y

YourKit 18

А

Анализ:
  дубликатов 390
  зависимостей 386
  кода 350

Б

Библиотека 300
  JavaDoc 298
  PLUS 750, 760
  глобальные 197, 294, 699
  модуля 293, 294
  проекта 294, 746
  экспорт 296

Г

Горячие клавиши 228

Ж

Живой шаблон 84, 85, 99, 208, 347, 613, 785
  настройка 210

З

Завершение кода 83, 166, 313, 329
  имя класса 84, 168, 333
  простое 83, 167, 329, 581
  умное 83, 168, 331
Закладка 48, 346
  нумерованная 50
  простая 48
Запуск:
  J2EE-приложения 546
  Java-приложения 538
  Junit-тестов 544
  апплета 540

И

Импорт 35, 257
  оптимизировать 99, 651
  статический 610, 617
Инспекция 86, 350, 769

К

Код:
  генерация 347
  информация 342
  проверка 350
  стиль 155
  форматирование 341
Команда:
  Active Tab
    Close 110
    Pin 110
  Analyze dependencies 86, 386
  Auto Indent Lines 85
  Breakpoint:
    Toggle Line 95
    Toggle Method 96
    View 96, 130, 551, 556

Call Hierarchy 69, 346
Close Editor:
  Active 34
  All 34
  All But Active 34
Close Project 34
Column Mode 43
Comment with:
  block comment 85
  line comment 85
Compile 90
Context Info 65, 342
Copy 42
Copy Path 42
Cut 42, 115
Debug 93, 116, 562
Delegate Methods 78
Delete 48
Edit Configurations 92, 515, 707
Error Description 66
Evaluate Expression 95, 130
  quick 95
Exit 41
Export Threads 96, 570
Export to HTML 36
Extermal JavaDoc 67
File Structure Popup 63
Find 50, 115
  in Path 59
  Next 52
  Previous 53
  Usages 54, 448
  Word at Caret 53
Full Screen 62
Generate 81, 317
  JavaDoc 99
Go To:
  Called Implementation 74
  Class 71, 320, 344
  Declaration 73, 320
  File 72, 345
  Implementation (s) 73, 345
  Last Edit Location 75
  Line 72
  Super Method 74
  Symbol 72, 345
  Type Declaration 74
Help Topics 110
Hide:
  Active Window 109
  All Windows 109
Hightlight Usages in File 59
Implement Methods 76, 328
Incremental Search 53
Insert Live Template 84
Inspect Code 86, 350
Jump to:
  Last Window 109
  Source 66
Locate Duplicates 86, 390
Macros 103
Make 513
  Module 90
  Project 90, 116
Method Hierarchy 69, 346
Migrate 104
Mode:
  Docked 109
  Floating 109
  Pinned 109
New 44, 312, 347, 400, 606, 619, 677
  Module 33, 310
  Project 32, 269
Open:
  File 33, 115, 723
  Project 33
Optimize Imports 99
Override Methods 75
Parameter Info 65, 342
Paste 42, 43, 115
Pop Frame 94, 130, 562—571
Print 38
Program:
  Pause 95, 130, 562
  Resume 95, 130, 567, 569
Quick:
  JavaDoc 64, 342
  Swith Scheme 68
Rebuild Project 89
Recent Files 67, 346
Redo 42, 115
Reformat Code 98, 341
Reopen 34
Replace 52
  in Path 61
Run 93, 116, 546
  to cursor 94, 130, 562, 566

Save All 22, 35, 115
Save File as Template 99
Select:
All 44
In 63
Tab 110
Set Up JSR014, 101
Settings 34, 116
Export 35
Import 35
Show:
Bookmarks 48
Execution Point 95, 572
Source 67
Whitespaces 62, 725
StatusBar 62
Step:
Force into 94, 130, 562—565
Into 94, 130, 562—565
Out 94, 130, 562, 565
Over 94, 130, 561—565
Stop 93, 130, 514
Surround With 79, 348
Live Template 85
Synchronize 36, 115
Template Project Settings 35
Toggle:
Bookmark 48
Breakpoint Enabled 96
ToolBar 62
Type Hierarchy 68, 346
Undo 42, 115
Компилятор 259, 512
javac 91, 262, 512
javac2, 91, 436, 686, 701
jikes 262, 512
RMI 514
фоновый режим 513
Компиляция 90
фоновый режим 89
форм 91
Контроль версий 623, 766
локальный. *См.* Local History

M

Меню 32
Analyze 32, 86
Build 32, 89
Code 32, 75
Edit 32, 42, 81
File 32, 143
Go To 32, 71
Help 32, 110
Refactor 32, 87, 442
Run 32, 92, 561, 572, 579, 707
Search 32, 50
Tools 32, 98, 626, 768
View 32, 62, 725
Window 32, 108, 739
Модуль 23, 155, 269
Ejb 272, 282, 302, 304, 583
J2EE-приложения 272, 286, 303, 305
Java 272
Web 272, 275, 299
зависимости 296
сборка 90
свойства 289, 299, 304, 305
создать 33, 272

H

Настройки 34, 153, 198, 256
прокси 248
IDE 154, 155
Appearance 155, 158
Application Servers 156, 250
Code Completion 155, 166
Colors&Fonts 155, 173
Debugger 155, 199, 563
Editor 155, 161
Errors 155, 170, 336
External Tools 156, 234
File Templates 156, 221
File Types 34, 156, 203
General 155, 156
Global Code Style 155, 179
Intention Power Pack 156, 255
JDK&Global Libraries 155, 195
Keymap 156, 228
Live Templates 156, 208
Local History 156, 207
Path Variables 156, 241
Plugins 156, 247
Resources 155, 202
TODO 156, 242

О

Окно:
Ant Build 136, 683
Commander 64, 124, 312, 458, 467, 624
CVS 137, 659
CVS File View 648
Debug 128, 549, 552, 569, 571, 576
Dependency Viewer 137, 386
Find 61, 125, 444, 448, 452, 461
Hierarchy 68, 133
Inspection 139
Messages 89, 120, 513, 514, 689
Project 63, 121, 124, 236, 274, 281, 291, 312, 320, 583, 600, 608, 624, 725, 740
Run 127, 544, 548
Structure 63, 64, 132, 447, 581, 740
TODO 131, 346
настройки 160
прозрачность 160
разделение 34
редактор 142
Отладка 549
выполнение по шагам 561
вычисление выражений 577
наблюдения 576

П

Панель:
инструментов 30, 62, 115, 727, 783
сравнения 179
статуса 141
Плагин 247, 248, 698, 784
настройка 247
песочница 702
репозиторий 698
Подсветка 334, 757
Подсказка 140, 255, 315, 318, 336, 337, 412, 770
Поиск:
быстрый 63—67
замена 52
в проекте 61
структурная 396
инкрементальный 53—58
в файле 58
класса 56
пакета 55
переменной 58
подсветка 58
поля класса 57
обращений 54
повторения 86
структурный 61, 392
Проект 23, 154, 155, 258, 783
закрыть 34
настройки 258
контроль версий 263, 631, 665, 672
пути 258, 289, 290
стиль кода 264
шаблон 35
открыть 33
пересобрать 89
собрать 90
создать 32, 269, 307
файл сборки Ant 690
управление 269

Р

Регистрация 114
Редактор 785
закрыть 34
режим выделения 43
Рефакторинг 87, 442, 783
введение константы 480
введение параметра 481
введение переменной 476
введение поля класса 477
выделение базового класса 485
выделение интерфейса 482
выделение метода 471
генерификация 508
замена временного использования 505
замена кода методом 474
замена конструктора методом фабрики 507
замена наследования делегированием 495
изменение сигнатуры 449
инкапсулировать поля 502
использовать интерфейс 488
конвертировать анонимный класс во внутренний 500
(*окончание рубрики см. на стр. 800)*

Рефакторинг *(окончание)*:
копирование 467, 468
обычный метод 457
переименование 443—449
директории 443
класса 446
локальной переменной 449
метода 446
пакета 444
параметров метода 448
полей класса 447
файла 444
перемещение 458—465
директории 459
класса 462
пакета 460
файла 459
члена класса 465, 490, 492
подстановка в код 497
статический метод 454
удаление 469
класс 469
метод 470
поле класса 470
Руководство
по продуктивности 112, 321

C

Сервер приложений. *См.* Application Server
Сообщество
разработчиков 698
Схема 68
сочетания клавиш 229
цвет и шрифт 174, 759

T

Точка останова 550
по исключению 556
при вызове метода 558
при доступе к полю класса 560
строковая 554

Ф

Файл:
BAT 28
BIN 26
CMD 205
DMG 27
EXE 26
IML 269, 722
IPR 33, 36, 269, 722
IWS 722
JAR 23
SH 28
tar.gz 27
кодировка 158
копировать путь 42
недавно
использованный 67, 320, 346
открытие 33
переход 72
печать 38
свойства 746
синхронизация 36, 156
сравнение 44, 158, 623
статус 179
типы 34
шаблон 47, 99, 221, 347
Форма 47, 91, 203, 398
Мастер связей 433
предварительный просмотр 406
создание 47, 399