Министерство образования Республики Беларусь

Учреждение образования «Гомельский государственный технический университет
имени П.О. Сухого»

Кафедра «Информационные технологии»

Курс лекций
старшего преподавателя Стефановского И.Л. по дисциплине

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

для студентов специальности 1-40 05 01
«Информационные системы и технологии (по направлениям)»,
дневной и заочной форм обучения

Гомель 2024

Введение

Учебная дисциплина «Технологии разработки программного обеспечения» является одной из дисциплин начального цикла подготовки студентов в области информационных технологий. Сущность учебной дисциплины составляют базовые принципы, методы и средства разработки программного обеспечения в части анализа и формализации требований, моделирования бизнес-процессов и алгоритмизации проектных решений, кодирования и отладки приложений.

Учебная дисциплина «Технологии разработки программного обеспечения» используется для освоения базового уровня моделирования, алгоритмизации и программирования решений профессиональных задач, способствует формированию интеллектуального и творческого потенциала личности будущего программиста.

Практическая деятельность инженера требует определенных знаний в области создания условий для обеспечения интеграции и сотрудничества ИТ-специалистов на всех этапах жизненного цикла разработки программного обеспечения, а также навыков применения социально-психологических методов управления, обладания позитивным профессиональным и личностным мышлением. Принципы и технологии создания дружественных пользовательских интерфейсов программного обеспечения, которые изучаются в ходе освоения студентами учебной дисциплины «Технологии разработки программного обеспечения», предполагают воспитание культуры и этики деловых отношений, развитие навыков разрешения конфликтных ситуаций, оптимизации морально-психологического климата в коллективе и поддержания партнерских взаимоотношений, направленных на творческое исполнение обязанностей.

Цель учебной дисциплины: формирование систематизированных знаний о жизненном цикле разработки программного обеспечения и технологиях, применяемых на различных его этапах, включая моделирование предметной области, формализацию требований, алгоритмизацию проектных решений, программную реализацию и отладку приложений.

Задачи учебной дисциплины:

- приобретение знаний о цели и основных задачах в области разработки программного обеспечения; приобретение

знаний об основах моделей и методологий жизненного цикла разработки программного обеспечения; приобретение знаний о парадигмах программирования;
- овладение базовыми методами анализа предметной области и формализации требований к разработке программного обеспечения;
- овладение базовыми методами моделирования и алгоритмизации для анализа и разработки проектных решений;
- овладение базовыми методами написания качественного и эффективного кода;
- изучение принципов юзабилити для создания дружественных пользовательских интерфейсов;
- ознакомление со стандартами разработки программных средств и систем и областью программной инженерии.

Учебная дисциплина «Технологии разработки программного обеспечения» является основой для такой учебной дисциплины, как «Объектно-ориентированное проектирование и программирование», также она может изучаться параллельно или после изучения учебной дисциплины «Основы алгоритмизации и программирования».

1 ВВЕДЕНИЕ В ТЕРМИНОЛОГИЮ И МЕТОДОЛОГИЮ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Жизненный цикл разработки программного обеспечения: этапы, модели и методологии

Разработка программного кода предваряется анализом и проектированием (первое означает создание функциональной модели будущей системы без учета реализации, для осознания программистами требований и ожиданий заказчика; второе означает предварительный макет, эскиз, план системы на бумаге). Трудозатраты на анализ и проектирование, а также форма представления их результатов сильно варьируются от видов проектов и предпочтений разработчиков и заказчиков.

Требуются также специальные усилия по организации процесса разработки. В общем виде это итеративно-инкрементальная модель, когда требуемая функциональность создается порциями, которые менеджеры и заказчик могут оценить, и тем самым есть возможность управления ходом разработки. Однако эта общая модель имеет множество модификаций и вариантов.

Разработку системы также необходимо выполнять с учетом удобств ее дальнейшего сопровождения, повторного использования и интеграции с другими системами. Это значит, что система разбивается на компоненты, удобные в разработке, годные для повторного использования и интеграции. А также имеющие необходимые характеристики по быстродействию. Для этих компонент тщательно прорабатываются интерфейсы. Сама же система документируется на многих уровнях, создаются правила оформления программного кода – то есть оставляются многочисленные семантические следы, помогающие создать и сохранить, поддерживать единую, стройную архитектуру, единообразный стиль, порядок.

Все эти и другие дополнительные виды деятельности, выполняемые в процессе промышленного программирования и необходимые для успешного выполнения заказов и будем называть **программной инженерией** (*software engineering*). Получается, что так мы обозначаем, во-первых, некоторую практическую деятельность, а во-вторых, специальную **область знания**. Или другими словами, научную дисциплину. Ведь для облегчения

выполнения каждого отдельного проекта, для возможности использовать разнообразный положительный опыт, достигнутый другими командами и разработчиками, этот самый опыт подвергается осмыслению, обобщению и надлежащему оформлению. Так появляются различные методы и практики (*best practices*) – тестирования, проектирования, работы над требованиями и пр., архитектурных шаблонов и пр. А также стандарты и методологии, касающиеся всего процесса в целом (например, *MSF*, *RUP*, *CMMI*, *Scrum*). Вот эти-то обобщения и входят в программную инженерию как в область знания. Необходимость в программной инженерии как в специальной области знаний была осознана мировым сообществом в конце 60-х годов прошлого века, более чем на 20 лет позже рождения самого программирования, если считать таковым знаменитый отчет фон Неймана "*First Draft of a Report on the EDVAC*", обнародованный им в 1945 году. Рождением программной инженерии является 1968 год – конференция *NATO Software Engineering*, г. Гармиш (ФРГ), которая целиком была посвящена рассмотрению этих вопросов. В сферу программной инженерии попадают все вопросы и темы, связанные с организацией и улучшением процесса разработки ПО, управлением коллективом разработчиков, разработкой и внедрением программных средств поддержки жизненного цикла разработки ПО. Программная инженерия использует достижения информатики, тесно связана с системотехникой, часто предваряется бизнес-реинжинирингом. Немного подробнее об этом контексте программной инженерии.

Информатика (*computer science*) – это свод теоретических наук, основанных на математике и посвященных формальным основам вычислимости. Сюда относят математическую логику, теорию грамматик, методы построения компиляторов, математические формальные методы, используемые в верификации и модельном тестировании и т.д. Трудно строго отделить программную инженерию от информатики, но в целом направленность этих дисциплин различна. Программная инженерия нацелена на решение проблем производства, информатика – на разработку формальных, математизированных подходов к программированию.

Системотехника (*system engineering*) объединяет различные инженерные дисциплины по разработке всевозможных искусственных систем – энергоустановок, телекоммуникационных систем, встроенных систем реального времени и т.д. Очень часто ПО

оказывается частью таких систем, выполняя задачу управления соответствующего оборудования. Такие системы называются *программно-аппаратными*, и участвуя в их создании, программисты вынуждены глубоко разбираться в особенностях соответствующей аппаратуры.

Бизнес-реинжиниринг (*business reengineering*) – в широком смысле обозначает модернизацию бизнеса в определенной компании, внедрение новых практик, поддерживаемых соответствующими новыми информационными системами. При этом акцент может быть, как на внутреннем переустройстве компании, так и на разработке нового клиентского сервиса (как правило, эти вопросы взаимосвязаны). Бизнес-реинжиниринг часто предваряет разработку и внедрение информационных систем на предприятии, так как требуется сначала навести определенный порядок в делопроизводстве, а лишь потом закрепить его информационной системой.

Связь программной инженерии (как области практической деятельности) с информатикой, системотехникой и бизнес-реинжинирингом показана на рис.1.1

Рис. 1.1. Взаимосвязи программной инженерии

Программное обеспечение

Будем понимать под **программным обеспечением** (ПО) множество развивающихся во времени логических предписаний, с помощью которых некоторый коллектив людей управляет и использует многопроцессорную и распределенную систему вычислительных устройств.

Это определение, данное Харальдом Милсом, известным специалистом в области программной инженерии из компании *IBM*, заключает в себе следующее:

– Логические предписания – это не только сами программы, но и различная документация (например, по эксплуатации программ) и шире – определенная система отношений между людьми, использующими эти программы в рамках некоторого процесса деятельности.

– Современное ПО предназначено, как правило, для одновременной работы со многими пользователями, которые могут быть значительно удалены друг от друга в физическом пространстве. Таким образом, вычислительная среда (персональные компьютеры, сервера и т.д.), в которой ПО функционирует, оказывается распределенной.

– Задачи решаемые современным ПО, часто требуют различных вычислительных ресурсов в силу различной специализации этих задач, из-за большого объема выполняемой работы, а также из соображений безопасности. Например, появляется сервер базы данных, сервер приложений и пр. Таким образом, вычислительная среда, в которой ПО функционирует, оказывается многопроцессорной.

– ПО развивается во времени – исправляются ошибки, добавляются новые функции, выпускаются новые версии, меняется его аппаратная база.

Свойства. Таким образом, ПО является сложной динамической системой, включающей в себя технические, психологические и социальные аспекты. ПО заметно отличается от других видов систем, создаваемых (созданных) человеком – механических, социальных, научных и пр., и имеет следующие особенности, выделенные Фредериком Бруксом в его знаменитой статье "Серебряной пули нет":

– Сложность программных объектов, которая существенно зависит от их размеров. Как правило, большее ПО (большее количество пользователей, больший объем обрабатываемых данных, более жесткие требования по быстродействию и пр.) с аналогичной функциональностью – это другое ПО. Классическая наука строила простые модели сложных явлений, и это удавалось, так как сложность не была характеристической чертой рассматриваемых явлений. (Сравнение программирования именно с наукой, а не с театром, кинематографом, спортом и другими областями человеческой

деятельности, оправдано, поскольку оно возникло, главным образом, из математики, а первые его плоды – программы – предназначались для использования при научных расчетах. Кроме того, большинство программистов имеют естественнонаучное, математическое или техническое образование. Таким образом, парадигмы научного мышления широко используются при программировании – явно или неявно.)

– Согласованность – ПО основывается не на объективных посылках (подобно тому, как различные системы в классической науке основываются на постулатах и аксиомах), а должно быть согласовано с большим количеством интерфейсов, с которыми впоследствии оно должно взаимодействовать. Эти интерфейсы плохо поддаются стандартизации, поскольку основываются на многочисленных и плохо формализуемых человеческих соглашениях.

– Изменяемость – ПО легко изменить и, как следствие, требования к нему постоянно меняются в процессе разработки. Это создает много дополнительных трудностей при его разработке и эволюции.

– Нематериальность – ПО невозможно увидеть, оно виртуально. Поэтому, например, трудно воспользоваться технологиями, основанными на предварительном создании чертежей, успешно используемыми в других промышленных областях (например, в строительстве, машиностроении). Там на чертежах в схематичном виде воспроизводятся геометрические формы создаваемых объектов. Когда объект создан, эти формы можно увидеть.

Процесс

Как мы работаем, какова последовательность наших шагов, каковы нормы и правила в поведении и работе, каков регламент отношений между членами команды, как проект взаимодействует с внешним миром и т.д.? Все это вместе мы склонны называть процессом. Его осознание, выстраивание и улучшение – основа любой эффективной групповой деятельности. Не случайно поэтому, что процесс оказался одним из основных понятий программной инженерии. Центральным объектом изучения программной инженерии является процесс создания

ПО – множество различных видов деятельности, методов, методик и шагов, используемых для разработки и эволюции ПО и связанных с ним продуктов (проектных планов, документации, программного кода, тестов, пользовательской документации и пр.).

Однако на сегодняшний день не существует универсального процесса разработки ПО – набора методик, правил и предписаний, подходящих для ПО любого вида, для любых компаний, для команд любой национальности. Каждый текущий процесс разработки, осуществляемый некоторой командой в рамках определенного проекта, имеет большое количество особенностей и индивидуальностей. Однако целесообразно перед началом проекта спланировать процесс работы, определив роли и обязанности в команде, рабочие продукты (промежуточные и финальные), порядок участия в их разработке членов команды и т.д. Будем называть это предварительное описание конкретным процессом, отличая его от плана работ, проектных спецификаций и пр. Например, в системе *Microsoft Visual Tem System* оказывается шаблон процесса, создаваемый или адаптируемый (в случае использования стандартного) перед началом разработки. В *VSTS* существуют заготовки для конкретных процессов на базе *CMMI*, *Scrum* и др.

В рамках компании возможна и полезна объединение и стандартизация всех текущих процессов, которую будем называть стандартным процессом. Последний, таким образом, оказывается некоторой базой данных, содержащей следующее:

– информацию, правила использования, документацию и инсталляционные пакеты средств разработки, используемых в проектах компании (систем версионного контроля, средств контроля ошибок, средств программирования – различных *IDE*, СУБД и т.д.);

– описание практик разработки – проектного менеджмента, правил работы с заказчиком и т.д.;

– шаблонов проектных документов – технических заданий, проектных спецификаций, тестовых планов и т.д. и пр.

Также возможна стандартизация процедуры разработки конкретного процесса как «вырезки» из стандартного. Основная идея стандартного процесса – курсирование внутри компании передового опыта, а также унификация средств разработки. Очень уж часто в компаниях различные департаменты и проекты сильно отличаются по зрелости процесса разработки, а также затруднено повторное использование передового опыта. Кроме того, случаются, что компания использует несколько средств параллельных инструментов разработки, например, СУБД средства версионного контроля. Иногда это бывает оправдано (например, таковы требования заказчика), часто

это необходимо – например, *Java .NET* (большая компетентность оффшорной компании позволяет ей брать более широкий спектр заказов). Но очень часто это произвольный выбор самих разработчиков.

В любом случае, такая множественность существенно затрудняет миграцию специалистов из проекта в проект, использование результатов одного проекта в другом и т.д. Однако при организации стандартного процесса необходимо следить, чтобы стандартный процесс не оказался всего лишь формальным, бюрократическим аппаратом. Понятие стандартного процесса введено и подробно описано в подходе *CMMI*.

Необходимо отметить, что наличие стандартного процесса свидетельствует о наличии «единой воли» в организации, существующей именно на уровне процесса. На уровне продаж, бухгалтерии и др. привычных для всех компаний процессов и активов единство осуществить не трудно. А вот на уровне процессов разработки очень часто каждый проект оказывается сам по себе (особенно в оффшорных проектах) – «текучка» захватывает и изолирует проекты друг от друга очень прочно.

В соответствии с базовым международным стандартом *ISO*/*IEC* 12207 все *процессы ЖЦ ПО* делятся на три группы:

1. **Основные процессы**:
   - приобретение;
   - поставка;
   - разработка;
   - эксплуатация;
   - сопровождение.
2. **Вспомогательные процессы**:
   - документирование;
   - управление конфигурацией;
   - обеспечение качества;
   - разрешение проблем;
   - аудит;
   - аттестация;
   - совместная оценка;
   - верификация.
3. **Организационные процессы**:
   - создание инфраструктуры;

– управление;
– обучение;
– усовершенствование.

В таблице 1.1 приведены ориентировочные описания основных процессов ЖЦ. Вспомогательные процессы предназначены для поддержки выполнения основных процессов, обеспечения качества проекта, организации верификации, проверки и тестирования ПО. Организационные процессы определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами.

Для поддержки практического применения стандарта *ISO*/*IEC* 12207 разработан ряд технологических документов: Руководство для *ISO*/*IEC* 12207 (*ISO*/*IEC TR* 15271:1998 *Information technology – Guide for ISO*/*IEC* 12207) и Руководство по применению *ISO*/*IEC* 12207 к управлению проектами (*ISO*/*IEC TR* 16326:1999 *Software engineering - Guide for the application of ISO*/*IEC* 12207 *to project management*).

Таблица 1.1

Содержание основных процессов ЖЦ ПО ИС (*ISO/IEC* 12207)

Процесс (исполнитель процесса)	**Действия**	**Вход**	**Результат**
Приобретение (заказчик)	– Инициирование – Подготовка заявочных предложений – Подготовка договора – Контроль деятельности поставщика – Приемка ИС	– Решение о начале работ по внедрению ИС – Результаты обследования деятельности заказчика – Результаты анализа рынка ИС/ тендера – План поставки/ разработки – Комплексный тест ИС	– Технико-экономическое обоснование внедрения ИС – Техническое задание на ИС – Договор на поставку/ разработку – Акты приемки этапов работы – Акт приемно-сдаточных испытаний
– Поставка	– Инициирование	– Техническое	– Решение об

(разработчик ИС)	– Ответ на заявочные предложения – Подготовка договора – Планирование исполнения – Поставка ИС	задание на ИС – Решение руководства об участии в разработке – Результаты тендера – Техническое задание на ИС – План управления проектом – Разработанная ИС и документация	участии в разработке – Коммерческие предложения/ конкурсная заявка – Договор на поставку/ разработку – План управления проектом – Реализация/ корректировка – Акт приемно-сдаточных испытаний
– Разработка (разработчик ИС)	– Подготовка – Анализ требований к ИС – Проектирование архитектуры ИС – Разработка требований к ПО – Проектирование архитектуры ПО – Детальное проектирование ПО – Кодирование и тестирование ПО – Интеграция ПО и квалификационное тестирование ПО – Интеграция ИС и квалификационное	– Техническое задание на ИС – Техническое задание на ИС, модель ЖЦ – Подсистемы ИС – Спецификации требования к компонентам ПО – Архитектура ПО – Материалы детального проектирования ПО – План интеграции ПО, тесты – Архитектура ИС, ПО,	– Используемая модель ЖЦ, стандарты разработки – План работ – Состав подсистем, компоненты оборудования – Спецификации требования к компонентам ПО – Состав компонентов ПО, интерфейсы с БД, план интеграции ПО – Проект БД, спецификации интерфейсов между

	тестирование ИС	документация на ИС, тесты	компонентами ПО, требования к тестам – Тексты модулей ПО, акты автономного тестирования – Оценка соответствия комплекса ПО требованиям ТЗ – Оценка соответствия ПО, БД, технического комплекса и комплекта документации требованиям ТЗ

Позднее был разработан и в 2002 г. опубликован стандарт на процессы *жизненного цикла* систем (*ISO*/*IEC* 15288 *System life cycle processes*). К разработке стандарта были привлечены специалисты различных областей: системной инженерии, программирования, управления качеством, человеческими ресурсами, безопасностью и пр. Был учтен практический опыт создания систем в правительственных, коммерческих, военных и академических организациях. Стандарт применим для широкого класса систем, но его основное предназначение – поддержка создания компьютеризированных систем.

Согласно стандарту *ISO*/*IEC* серии 15288 в структуру ЖЦ следует включать следующие группы процессов:

1. **Договорные процессы**:
   - приобретение (внутренние решения или решения внешнего поставщика);
   - поставка (внутренние решения или решения внешнего поставщика).

2. **Процессы предприятия**:
   - управление окружающей средой предприятия;
   - инвестиционное управление;
   - управление ЖЦ ИС;
   - управление ресурсами;
   - управление качеством.
3. **Проектные процессы**:
   - планирование проекта;
   - оценка проекта;
   - контроль проекта;
   - управление рисками;
   - управление конфигурацией;
   - управление информационными потоками;
   - принятие решений.
4. **Технические процессы**:
   - определение требований;
   - анализ требований;
   - разработка архитектуры;
   - внедрение;
   - интеграция;
   - верификация;
   - переход;
   - аттестация;
   - эксплуатация;
   - сопровождение;
   - утилизация.
5. **Специальные процессы**:
   - определение и установка взаимосвязей исходя из задач и целей.

Одним из базовых понятий методологии проектирования ИС является понятие жизненного цикла ее программного обеспечения (ЖЦ ПО). ЖЦ ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт *ISO*/*IEC* 12207 [1] (*ISO* – *International Organization of Standardization* – Международная

организация по стандартизации, *IEC – International Electrotechnical Commission* – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Структура ЖЦ ПО по стандарту *ISO*/*IEC* 12207 базируется на трех группах процессов:

– основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);

– вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);

– организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации,

проверки и тестирования ПО. Верификация – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта *ISO* 12207-2 [2].

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде *жизненного цикла* (ЖЦ) ИС, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ ИС позволяет

спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл ИС можно представить, как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. *Модель жизненного цикла* – структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

В настоящее время известны и используются следующие *модели жизненного цикла*:

– ***Каскадная модель*** (рис. 1.2) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.

– ***Поэтапная модель с промежуточным контролем*** (рис. 1.3). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.

– ***Спиральная модель*** (рис. 1.4). На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (макетирования).

Рис. 1.2. Каскадная модель ЖЦ ИС

Рис. 1.3. Поэтапная модель с промежуточным контролем

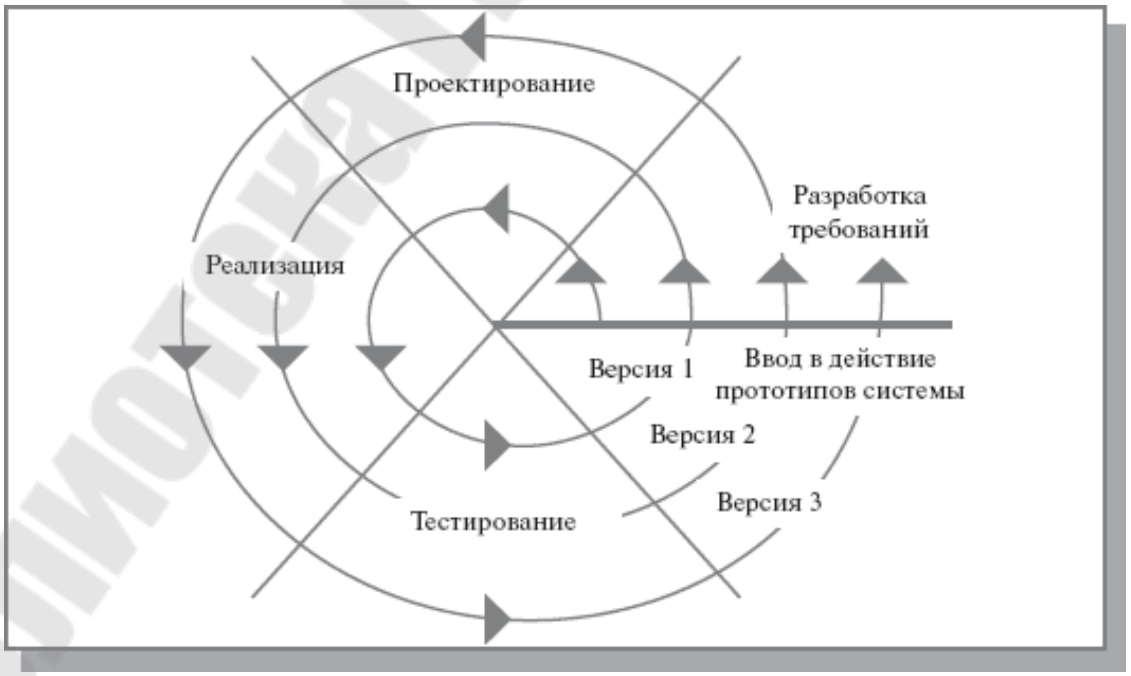

Рис. 1.4. Спиральная модель ЖЦ ИС

На практике наибольшее распространение получили две основные *модели жизненного цикла*:

– *каскадная модель* (характерна для периода 1970-1985 гг.);

– *спиральная модель* (характерна для периода после 1986.г.).

В ранних проектах достаточно простых ИС каждое приложение представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

Можно выделить следующие положительные стороны применения каскадного подхода:

– на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;

– выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ИС оказывается соответствующим *поэтапной модели с промежуточным контролем*.

Однако и эта схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ИС зафиксированы в виде технического задания на все время ее создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

Спиральная модель

Процесс итеративной (или инкрементальной) разработки стал эволюционным развитием модели водопада. ЖЦ была предложена для преодоления перечисленных проблем. На этапах анализа и

проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов *жизненного цикла*, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Несмотря на настойчивые рекомендации компаний - вендоров и экспертов в области проектирования и разработки ИС, многие компании продолжают использовать *каскадную модель* вместо какого-либо варианта итерационной модели. Основные причины, по которым *каскадная модель* сохраняет свою популярность, следующие:

1. **Привычка** – многие ИТ-специалисты получали образование в то время, когда изучалась только *каскадная модель*, поэтому она используется ими и в наши дни.

2. **Иллюзия снижения рисков** участников проекта (заказчика и исполнителя). *Каскадная модель* предполагает разработку законченных продуктов на каждом этапе: технического задания, технического проекта, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом

окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование *каскадной модели* создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки (*fixed price*). Второй тип предполагает повременную оплату работы (*time work*). Выбор того или иного типа контракта зависит от степени определенности задачи. *Каскадная модель* с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, и, следовательно, *каскадная модель* разработки и внедрения. *Спиральная модель* чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.

Проблемы внедрения при использовании итерационной модели. В некоторых областях *спиральная модель* не может применяться, поскольку невозможно использование/тестирование продукта, обладающего неполной функциональностью (например, военные разработки, атомная энергетика и т.д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, учетной политики, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя

указанные сложности, заказчики выбирают *каскадную модель*, чтобы "внедрять систему один раз".

Гибкие методологии

В течение 1990-х годов все больше разработчиков ПО начинали искать альтернативу традиционным, как правило, основанным на модели водопада, процессам разработки. К 2000 году существовало уже целое множество так называемых легковесных (*lightweight*) методологий. (Я использую термин «методология», а не «процесс», поскольку гибкие методологии включают в себя множество практик и технологий, выходящих за рамки описания процессов.)

В 2001 году группа создателей и экспертов по различным легковесным методологиям провела семинар, на котором были сформулированы основные принципы гибкой разработки ПО (так называемый *Agile Manifesto*). На том же семинаре было предложено новое название легковесных методологий – гибкая разработка (*agile software development*).

Общими особенностями гибких методологий являются:

– Ориентированность на людей – как разработчиков, так и заказчиков. Считается, что умение собрать в проектной команде «правильных» людей определяет успех или неудачу проекта в значительно большей степени, чем любые процессы или технологии.

– Использование устных обсуждений вместо формальных спецификаций везде, где это возможно. Обсуждения должны быть главным способом коммуникации как с заказчиком, так и внутри проектной команды.

– Итеративная разработка с возможно более короткой (в разумных пределах) продолжительностью итерации, при этом в результате каждой итерации выпускается полноценная работающая версия продукта.

– Ожидание изменений – в гибком процессе проектная команда не пытается зафиксировать требования в начале проекта и затем следовать жестко определенному плану. Изменения могут быть сделаны на сколь угодно позднем этапе проекта.

По всей видимости, из методологий гибкой разработки самое широкое распространение получило экстремальное программирование (*eXtreme Programming*, *XP*), поэтому именно его мы рассмотрим подробнее.

XP

Методология *XP* была создана Кентом Беком (*Kent Beck*) в 1996 году в ходе попытки спасти провальный проект по разработке системы расчета зарплаты для компании Крайслер. В 2000 году проект был закрыт, но *XP* к тому времени уже получила известность и начала распространяться среди разработчиков ПО.

XP наследует все общие принципы гибких методологий, достигая их при помощи двенадцати инженерных практик. Ниже описаны самые интересные из специфических технологий и практик *XP*:

– В проектной команде должен постоянно работать так называемый представитель заказчика – он обладает детальной информацией о необходимой функциональности, определяет приоритеты отдельных требований, оценивает качество создаваемой системы. Технически, представитель заказчика может быть и сотрудником фирмы разработчика – менеджером продукта, бизнес-аналитиком и т.п.

– Пользовательские истории – короткие неформальные описания прецедентов использования системы. В *XP* истории являются основным и, вместе с приемочными тестами, единственным средством спецификации требований. Поскольку истории очень лаконичны, участникам проекта обычно требуются более детальная информация по функциональности системы – они получают ее непосредственно от представителя заказчика.

– Разработка через тестирование (*test driven development*) – в *XP* становится особенно важным, чтобы весь создаваемый код был покрыт автоматическими юнит-тестами (почему это так, станет понятно дальше). Этого можно добиться при помощи простого правила – новый код может быть написан исключительно для того, чтобы увеличить число успешно проходящих юнит-тестов. Фактически это означает, что перед реализацией новой функции разработчик должен создать соответствующий тест, а написание кода должно завершиться в тот момент, когда начнет проходить новый, а также все существующие тесты. Если после этого окажется, что новая функция реализована не полностью, необходимо создать еще один тест и повторить весь цикл заново.

– Архитектура системы должна быть максимально простой. *XP* не рекомендует проектировать в расчете на будущее развитие системы; идеальная архитектура должна не более чем поддерживать существующую функциональность. Цель такого минималистичного

подхода к проектированию – избежать бесполезных инвестиций в архитектурные решения, которые часто оказываются выброшенными после очередного изменения требований. Вместо этого архитектура постоянно изменяется и развивается вместе с системой.

– Постоянное изменение архитектуры требует постоянной переработки и улучшения кода – рефакторинга. В *XP* поощряется коллективное владение кодом – увидев возможность улучшения в любом компоненте системы, разработчик может провести необходимые рефакторинги вне зависимости от того, кто является основным разработчиком компонента. Возможные ошибки, внесенные рефакторингом, должны быть тут же обнаружены автоматическими тестами.

– Все изменения, сделанные разработчиками, после автоматического тестирования практически сразу попадают в основной репозиторий. Таким образом этап интеграции как таковой отсутствует или, что-то же самое, происходит постоянно. *XP* называет эту практику непрерывной интеграцией.

– Парное программирование – наверное, самая противоречивая практика *XP*. Использование парного программирования не означает, что на двух разработчиков в организации должен быть выделен только один компьютер, однако большая часть написания кода должна проходить в парах. Считается, что при этом общая эффективность разработки повышается за счет более продуманых решений, меньшего количества ошибок, тщательного написания юнит тестов и т.п.

– Продолжительность рабочей недели не должна превышать 40 часов. По сравнению с обычной практикой постоянных переработок, в средне- и долгосрочной перспективе это повышает производительность проектной команды за счет уменьшения стресса и переутомления.

Жизненный цикл проекта *XP*

Жизненный цикл проекта в *XP* состоит из последовательности релизов. Каждый релиз – это полноценная версия продукта, которую может использовать заказчик, и содержащая дополнительную функциональность по сравнению с предыдущим релизом. Релиз появляется в результате одной или нескольких итераций, длящихся от одной до четырех недель.

В *XP* не рекомендуется тратить много времени на планирование; сам процесс планирования называется игрой (*planning game*).

Подробный план составляется только на очередную итерацию и ближайшие один-два релиза.

Планирование релиза состоит из следующих шагов:

– Заказчик формулирует свои требования в виде историй, которые оцениваются по трудоемкости разработчиками. Оценки трудоемкости делаются в так называемых идеальных днях – времени, которое некий воображаемый разработчик потратит на реализацию истории при полном отсутствии отвлекающих факторов, параллельных задач, перерывов и т.п.

– Истории сортируются по приоритету, рискам, сложности реализации.

– Определяется фактическая производительность команды (какое число реальных дней соответствует идеальному дню).

– На основании фактической производительности определяется, какие истории войдут в очередной релиз и за какое время он будет завершен.

– Итерация планируется аналогичным образом.

– Предварительно определяется набор историй для итерации.

– Истории разбиваются на задачи (*tasks*), которые распределяются между разработчиками. В отличие от историй, которые описывают поведение системы с точки зрения пользователя, задачи составляются на техническом уровне, например «модифицировать схему базы данных» или «провести рефакторинг».

– Разработчики оценивают свои задачи. Казалось бы, это ненужное повторение операции оценки историй, однако на этом этапе уточненные оценки должны быть более реалистичными. Поскольку задачи теперь оценивают их непосредственные исполнители, в оценках учитывается индивидуальная производительность, зависящая от опыта, знакомства с используемыми технологиями и т.п.

– На основе уточненных оценок задач подсчитывается общая загрузка команды. В зависимости от загрузки некоторые истории могут быть перенесены на следующую итерацию или, наоборот, добавлены в текущую.

– Периодически в ходе проекта измеряется фактическая производительность команды. Если она начинает сильно отличаться от значения, которое было использовано при планировании, график выхода релизов должен быть пересмотрен.

Практика

Во многом *XP* была создана как попытка описать процессы и практики, которые часто как бы сами собой появляются в эффективных, сплоченных командах разработчиков. Может показаться, что процесс в таких командах отсутствует, и, скорее всего, то же самое будут утверждать сами разработчики. Однако это не так, поскольку работа профессиональной команды всегда четко (хотя и неформально) организована и не имеет ничего общего с беспорядком и хаосом.

Это во многом определяет условия, необходимые для эффективного использования *XP*. Прежде всего, *XP* имеет шансы работать только в команде опытных, профессиональных разработчиков. Поскольку большую роль в *XP* играет прямое общение, команда не должна быть разбита на несколько частей – внедрение *XP* в распределенной географически команде будет крайне рискованным мероприятием. По той же причине возможный размер команды ограничен сверху – по всей видимости, числом в 10-15 человек.

Другие практики *XP* приносят свои ограничения. Далеко не всегда можно обеспечить постоянное присутствие представителя заказчика в проектной команде (например, если потенциальные пользователи системы делятся на несколько классов с частично конкурирующими требованиями).

Поскольку *XP* практически не делает попыток предотвратить размывание границ проекта (*scope creep*), будет не очень хорошей идеей использовать *XP* в проекте с фиксированной ценой. Фактически проекты *XP* обладают жестким графиком, но переменными границами, поэтому предпочтительным типом контракта будет повременная оплата (*time&materials*).

Практика поддержания максимально простой архитектуры может завести в тупик в конце проекта, когда окажется, что для реализации завершающих историй требуется полное перепроектирование системы (оказывается, нам нужно было предусмотреть возможность интеграции с системой *SAP R*/3, а также перевода на японский язык всего пользовательского интерфейса!). Даже если подобная ситуация не возникнет, нет никакой гарантии, что создаваемая *ad hoc* архитектура не будет намного более запутанной и сложной, чем было бы продуманное заранее решение.

Подобным образом может неконтролируемо откладываться разрешение некоторых нетехнологических рисков, наподобие неопределенных границ проекта.

Несмотря на все перечисленные ограничения, *XP* может замечательно работать в подходящих для него условиях. Благодаря крайне низким накладным расходам, в таких ситуациях этот процесс может показать исключительную эффективность.

XP является достаточно гибкой методологией. Не обязательно внедрять *XP* во всей компании, вполне разумно ограничиться теми командами и проектами, которые могут получить от этого реальный выигрыш. Например, для разработки ядра продукта можно использовать *XP*, а проекты по внедрению основывать на процессе *RUP*. Также не обязательно использовать все классические практики *XP* (на самом деле, мало кто это делает) – как правило, разумно ограничиться теми из них, которые сочетаются с корпоративной культурой и особенностями проектов.

CMMI

Модель *Capability Maturity Model Integration* (*CMMI*) была разработана в течение 1990-х годов в университете Карнеги-Меллона (*Carnegie Mellon University*.) совместно с *Software Engineering Institute* (*SEI*) и другими организациями. Одним из главных спонсоров разработки стало Министерство обороны США. *CMMI* была создана путем объединения (отсюда слово *Integration* в названии) трех более ранних специализированных моделей разработки: *CMM for Software* (*SW-CMM*), *Electronic Industries Alliance Interim Standard* (*EIA/IS*) 731 и *Integrated Product Development Capability Maturity Model* (*IPD-CMM*). Последняя версия спецификации *CMMI* 1.2 была опубликована в августе 2006 года.

Цель внедрения *CMMI* – построить инфраструктуру процессов, устанавливающую общий стандарт выполнения проектов внутри организации. Для каждого отдельного проекта стандартный процесс должен быть модифицирован в соответствии со спецификой этого проекта. При помощи формальных метрик измеряется эффективность процессов, а сами процессы постоянно оптимизируются.

Необходимо сказать, что *CMMI* не описывает какой-то конкретный процесс разработки. *CMMI*-совместимым может быть проект с водопадным, итеративным или другим жизненным циклом. Правильнее думать о *CMMI* как о наборе элементов процессов,

приемов и методик, из которых, как из конструктора, нужно собрать законченный процесс.

Внедрение *CMMI* в организации может происходить на непрерывной (*continuous*) или ступенчатой (*staged*) основе. Содержание модели *CMMI* в обоих случаях одно и тоже, меняется только ее представление. Непрерывное представление дает возможность производить улучшения в отдельных процессных областях в произвольном порядке. Ступенчатое представление определяет четкую последовательность шагов по внедрению *CMMI*; каждый шаг соответствует достижению так называемого уровня зрелости (*maturity level*). Организации необходимо принять решение, до какого уровня зрелости она намерена дойти, а также необходимо ли проходить официальную сертификацию на соответствие этому уровню.

Остановимся подробнее на ступенчатом представлении, поскольку именно оно чаще всего используется на практике.

Структура *CMMI* (ступенчатое представление)

Основными элементами модели *CMMI* являются процессные области, общие и специальные задачи, общие и специальные практики.

CMMI определяет 22 процессные области, такие как планирование проекта (*Project Planning*), управление рисками (*Risk Management*), разработка требований (*Requirements Development*) и т.д. В ступенчатом представлении процессные области сгруппированы по пяти уровням зрелости (от 1 до 5). В непрерывном представлении каждая процессная область находится на одном из шести (от 0 до 5) уровней производительности (*capability level*).

Процессы в каждой процессной области должны достигать ряда целей. Общие цели (*generic goals*) относятся к нескольким процессным областям. Специальные цели (*specific goals*) уникальны для своей процессной области.

Для достижения специальных и общих целей служат специальные и общие практики (*practices*). Практики – это деятельность или задачи, которые должны быть выполнены для достижения соответствующей цели.

Рис. 1.5. Иерархия целей

В качестве примера рассмотрим процессную область Планирование проекта (*Project Planning*). В нее входит определение проектных артефактов, разбиение работ на отдельные задачи и их оценка, планирование необходимых ресурсов, составление графика проекта, анализ рисков, и т.д. В результате планирования проекта создается план проекта (*project plan*) – основной документ для организации и контроля проектных работ, управления бюджетом и оценки доходности, управления изменениями и рисками.

ПРИМЕЧАНИЕ

Часто путают план (*plan*) и график (*schedule*) проекта. План проекта – более широкое понятие; как правило, он включает в себя график.

Процессная область

Планирование проекта должна достигать трех специальных и двух общих целей. В рамках этих целей необходимо:

– Установить оценки (специальная цель): подготовить реалистичные оценки, такие как трудоемкость и стоимость проекта, для дальнейшего планирования.

– Разработать проектный план (специальная цель): создать документ, одобренный заинтересованными сторонами, описывающий жизненный цикл проекта, бюджет, ресурсы, риски, график, стратегию обеспечения качества и т.п.

– Получить обязательства участников проекта (специальная цель): участники проекта, ответственные за его выполнение, должны считать проектный план реалистичным и выполнимым, и подтвердить обязательство выполнить свои задачи в рамках заданного графика, ресурсов и качества.

– Учредить управляемый процесс (общая цель): общее требование к процессам на уровне зрелости 2.

– Учредить определенный (*defined*) процесс (общая цель): общее требование к процессам на уровне зрелости 3.

Перечислим практики, позволяющие достичь цели «разработать проектный план»:

– Установить бюджет и график проекта.

– Идентифицировать риски.

– Спланировать управление данными (определить источники и форматы данных, необходимые процедуры для миграции, репликации и получения данных, требования по безопасности и т.д.).

– Определить необходимые ресурсы.

– Определить требования к профессиональным навыкам сотрудников, запланировать тренинги.

– Запланировать вовлечение всех заинтересованных лиц.

– Создать проектный план.

Как уже говорилось, ступенчатое представление *CMMI* позволяет классифицировать организации по уровням зрелости:

– Уровень 1: начальный (*Initial*). Процессы в организации формируются спонтанно и хаотично. Отдельные проекты могут завершаться успешно, однако вероятность их успешного завершения, а также соответствие запланированным графику и бюджету малопредсказуемы.

– Уровень 2: управляемый (*Managed*). Основная задача при внедрении этого уровня – установить для каждого проекта стандартные процессы управления требованиями, планирования, наблюдения и контроля над проектом, управления конфигурациями. Естественно, процессы должны соответствовать требованиям *CMMI*, а именно достигать всех специальных целей в соответствующих процессных областях, а также общих целей, относящихся к уровню 2. Для каждого проекта периодически проводится анализ его соответствия установленным процедурам и, при необходимости,

корректирование процессов. Также периодически измеряются и анализируются метрики (производительности, качества и т.п.)

– Уровень 3: определенный (*Defined*). Уровень 3 включает в себя все требования уровня 2, а также добавляет множество обязательных процессных областей (разработка требований, интеграция, тестирование, управление рисками, и другие). Однако главное его отличие от уровня 2 заключается не в этом. В то время как уровень 2 требует определить процесс для каждого отдельного проекта, на уровне 3 набор стандартных процессов должен существовать на уровне всей организации. Процессы для отдельных проектов создаются при помощи настройки (*tailoring*) стандартных процессов организации, при этом изменения должны быть ограничены правилами настройки (*tailoring guidelines*). Настройка процессов – это также процессная область, определенная в *CMMI* как *Organizational Process Definition*.

– Уровень 4: количественно управляемый (*Quantitatively Managed*). Этот уровень требует количественного измерения метрик производительности и качества используемых процессов. По сравнению с уровнем 3 уровень 4 дает возможность предсказания и сравнения (на основе статистических данных) измеряемых характеристик процессов.

– Уровень 5: оптимизирующий (*Optimizing*). Уровень 5 – высшая ступень развития организации, использующей *CMMI*. При помощи метрик с уровня 4 организация постоянно изменяет свои процессы для того, чтобы улучшить производительность и качество создаваемых продуктов.

Практика

В каких организациях и на каких проектах лучше всего работает *CMMI* ? Для того чтобы понять это, необходимо выделить основные характеристики модели.

CMMI вполне заслуженно считается тяжеловесной методологией, причем требует ощутимых затрат как вовремя, так и после внедрения. Тяжеловесность процессов нарастает с продвижением по уровням зрелости. Уже первый из интересных в практическом смысле уровней зрелости – уровень 3 – требует значительных усилий по обучению проектной команды, ведению проектной документации, периодическому аудиту процессов.

Как уже говорилось, под требования *CMMI* можно подогнать самые разнообразные процессы разработки, однако некоторые

сочетания явно не имеют смысла. К примеру, вряд ли можно найти разумное применение процессу *XP*, дополненному формальным документированием требований и сбором метрик.

Что получает организация в обмен на усилия и ресурсы, потраченные на внедрение *CMMI* ? Прежде всего, предсказуемость сроков и бюджета выполнения более или менее аналогичных проектов. Точность планирования – это основная цель 3 и 4 уровней зрелости *CMMI*, эффективность разработки становится ключевым фактором лишь на уровне 5. Благодаря хорошо документированным процессам и промежуточным результатам работ становится возможным быстро подключать к проекту новых сотрудников (возможно, более типичная ситуация – заменять ушедших).

Таким образом, представляется целесообразным использование *CMMI* в следующих условиях:

– В относительно больших организациях, которые могут позволить себе значительные накладные расходы на внедрение и использование процесса.

– При высокой текучести кадров, когда тем не менее необходимо поддерживать скорость разработки и качество продуктов на достойном уровне.

– В случаях, когда организация выполняет большое количество более или менее однотипных проектов, *CMMI* позволяет достаточно точно планировать сроки и бюджет, и выполнять проекты в этих рамках.

– Важная, а часто и определяющая причина для внедрения *CMMI* – возможность официальной сертификации на достижение определенного уровня зрелости. Нередко наличие или отсутствие сертификации по *CMMI* является решающим фактором в выборе компании-подрядчика.

Очевидно, что не имеет большого смысла использовать *CMMI* для одного или нескольких отдельных проектов. Внедрение должно происходить во всей организации, департаменте и т.д.

Вряд ли использование *CMMI* принесет какие-либо преимущества при разработке новых продуктов, а особенно в исследовательских проектах (*research and development*). Поскольку при этом отнюдь не исчезают все накладные расходы, связанные с этой моделью, на мой взгляд, нецелесообразно использовать *CMMI* для разработки новых продуктов или сервисов. Однако проекты по

внедрению этих продуктов вполне могут эффективно использовать *CMMI*.

Выводы

Мы увидели, что современные процессы, использующиеся в реальных проектах, весьма разнообразны. Каждый из них имеет свои преимущества, которые проявляются в соответствующих условиях. Даже, казалось бы, безнадежно устаревшая водопадная модель совершенна адекватна для некоторых проектов. Каждый процесс обладает также и рядом характеристик, которые ограничивают область его эффективного использования. Эта ситуация вполне типична для разработки ПО, где уже накоплено множество технологий и методик, но не существует универсального метода, оптимального для любой задачи.

Несомненно, область процессной инженерии еще далека от зрелости и в ближайшем будущем будут созданы новые методологии. Из существующих процессов, имеет смысл обратить внимание на методологию *Microsoft Solutions Framework* (*MSF*). Это гибкая и достаточно легковесная методология, построенная на итеративной модели разработки. Привлекательной особенностью *MSF* является большое внимание к созданию эффективной и небюрократизированной проектной команды. Для достижения этой цели *MSF* предлагает достаточно нетрадиционные подходы к организационной структуре, распределению ответственности и принципам взаимодействия внутри команды.

Интересным примером является *OpenUP* – семейство открытых (в отличие от проприетарного *RUP*) процессов, создаваемое в рамках проекта *Eclipse Process Framework* (*EPF*). Процесс *OpenUP*/*Basic* основан на принципах *RUP*, максимально упрощенного для гибкой разработки небольших проектов. Одновременно с ним развивается *OpenUP*/*MDD*, следующий методологии разработки через моделирование (*Model Driven Development*).

OpenUP/*Basic* и *OpenUP*/*MDD* реализуются в виде расширений *EPF Composer*, средства для создания и настройки процессов. Пользователи *EPF Composer* могут создавать свои процессы, используя в качестве компонентов фрагменты процессов, доступные в расширениях наподобие *OpenUP*/*Basic* и *OpenUP*/*MDD*.

RUP

Один из самых известных процессов, использующих итеративную модель разработки – *Rational Unified Process* (*RUP*). Он

был создан во второй половине 1990-х годов в компании *Rational Software*. Основными разработчиками были Филипп Крачтен (*Philippe Kruchten*), Грейди Буч (*Grady Booch*), Джеймс Рамбо (*James Rumbaugh*) и Айвар Якобсон (*Ivar Jacobson*). Кстати, последние трое являются также создателями нотации *UML.*

Термин *RUP* означает как методологию разработки, так и продукт компании *IBM* (ранее – *Rational*) для управления процессами разработки. Методология *RUP* описывает абстрактный общий процесс, на основе которого организация или проектная команда должна создать специализированный процесс, ориентированный на ее потребности.

Можно выделить следующие основные характеристики процесса *RUP*.

Разработка требований

Для описания требований в *RUP* используются прецеденты использования (*use cases*). Это не слишком удивительно, учитывая, что один из создателей *RUP*, Айвар Якобсон, является также автором концепции прецедента использования. Полный набор прецедентов использования системы вместе с логическими отношениями между ними (прецеденты могут включать и расширять другие прецеденты) называется моделью прецедентов использования.

Каждый прецедент использования – это описание сценария взаимодействия пользователя с системой, полностью выполняющего конкретную пользовательскую задачу. Разумеется, не имеет смысла документировать в виде прецедентов нефункциональные требования (к производительности, качеству т.д.). Однако согласно *RUP* все функциональные требования должны быть представлены в виде прецедентов использования. Считается, что модель прецедентов дает более целостное представление о функциональности системы по сравнению с традиционным описанием требований (перечислением функций, которыми должна обладать система).

Итеративная разработка

Проект в *RUP* состоит из последовательности итераций с рекомендованной продолжительностью от 2 до 6 недель. Основной единицей планирования итераций является прецедент использования. Перед началом очередной итерации определяется набор прецедентов использования, которые будут реализованы к ее завершению.

Итеративная модель, подробно описанная выше, позволяет вносить необходимые изменения в требования, проектные решения и реализацию в ходе проекта.

Архитектура

Можно сказать, что *RUP* – ориентированная на архитектуру методология. Считается, что реализация и тестирование архитектуры системы должны начинаться на самых ранних стадиях проекта. *RUP* использует понятие исполняемой архитектуры (*executable architecture*) – основы приложения, позволяющей реализовать архитектурно значимые прецеденты использования. Основы исполняемой архитектуры должны быть реализованы как можно раньше. Это позволяет оценить адекватность принятых архитектурных решений и внести необходимые коррективы еще в начале проекта. Таким образом, для первых нескольких итераций необходимо выбирать прецеденты, которые требуют реализации большей части архитектурных компонентов.

RUP поощряет использование визуальных средств для анализа и проектирования. Как правило, используется нотация и, соответственно, средства моделирования *UML* (такие как *Rational Rose*). Модель предметной области документируется в виде диаграммы классов, модель прецедентов использования – при помощи диаграммы прецедентов, взаимодействие компонентов системы между собой описывается диаграммой последовательности и т д.

Жизненный цикл проекта

Жизненный цикл проекта *RUP* состоит из четырех фаз. Последовательность этих фаз фиксирована, но число итераций, необходимых для завершения каждой фазы, определяется индивидуально для каждого конкретного проекта. Фазы *RUP* нельзя отождествлять с фазами водопадной модели– их назначение и содержание принципиально различны.

Начало (*Inception*)

Фаза Начало обычно состоит из одной итерации. В ходе выполнения этой фазы необходимо:

– Определить видение и границы проекта.

– Создать экономическое обоснование (*business case*).

– Идентифицировать большую часть прецедентов использования и подробно описать несколько ключевых прецедентов.

– Найти хотя бы одно возможное архитектурное решение.

– Оценить бюджет, график и риски проекта.

Если после завершения первой итерации заинтересованные лица приходят к выводу о целесообразности выполнения проекта, проект переходит в следующую фазу. В противном случае проект может быть отменен или проведена еще одна итерация фазы Начало.

Проектирование (*Elaboration*)

В результате выполнения этой фазы на основе требований и рисков проекта создается основа архитектуры системы. Проектирование может занимать до двух-трех итераций или быть полностью пропущенным (если в проекте используется архитектура существующей системы без изменений). Целями этой фазы являются:

– Детальное описание большей части прецедентов использования.

– Создание оттестированной (при помощи архитектурно значимых прецедентов использования) базовой архитектуры.

– Снижение основных рисков и уточнение бюджета и графика проекта.

В отличие от модели водопада, основным результатом этой фазы является не множество документов со спецификациями, а действующая система с 20-30% реализованных прецедентов использования.

Построение (*Construction*)

В этой фазе (длящейся от двух до четырех итераций) происходит разработка окончательного продукта. Во время ее выполнения создается основная часть исходного кода системы и выпускаются промежуточные демонстрационные прототипы.

Внедрение (*Transition*)

Целями фазы Внедрения являются проведение бета-тестирования и тренингов пользователей, исправление обнаруженных дефектов, развертывание системы на рабочей площадке, при необходимости – миграция данных. Кроме того, на этой фазе выполняются задачи, необходимые для проведения маркетинга и продаж.

Фаза внедрения занимает от одной до трех итераций. После ее завершения проводится анализ результатов выполнения всего проекта: что можно изменить для улучшения эффективности в будущих проектах ?

Каждая из стадий создания системы предусматривает выполнение определенного объема работ, которые представляются в

виде *процессов ЖЦ*. *Процесс* определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

Существует целый ряд стандартов, регламентирующих ЖЦ ПО, а в некоторых случаях и процессы разработки.

Значительный вклад в теорию проектирования и разработки информационных систем внесла компания *IBM*, предложив еще в середине 1970-х годов методологию *BSP* (*Business System Planning* - методология организационного планирования). Метод структурирования информации с использованием матриц пересечения бизнес-процессов, функциональных подразделений, функций систем обработки данных (информационных систем), информационных объектов, документов и баз данных, предложенный в *BSP*, используется сегодня не только в ИТ-проектах, но и проектах по реинжинирингу бизнес-процессов, изменению организационной структуры. Важнейшие шаги процесса *BSP*, их последовательность (получить поддержку высшего руководства, определить процессы предприятия, определить классы данных, провести интервью, обработать и организовать данные интервью) можно встретить практически во всех формальных методиках, а также в проектах, реализуемых на практике.

Среди наиболее известных стандартов можно выделить следующие:

– ГОСТ 34.601-90 - распространяется на автоматизированные системы и устанавливает стадии и этапы их создания. Кроме того, в стандарте содержится описание содержания работ на каждом этапе. Стадии и этапы работы, закрепленные в стандарте, в большей степени соответствуют *каскадной модели* жизненного цикла [3].

– *ISO*/*IEC* 12207:1995 - стандарт на процессы и организацию *жизненного цикла*. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов [4].

– *Custom Development Method* (методика *Oracle*) по разработке прикладных информационных систем - технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением *Oracle*. Применяется *CDM* для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий "быстрой

разработки" (*Fast Track*) или "облегченного подхода", рекомендуемых в случае малых проектов.

– *Rational Unified Process* (*RUP*) предлагает итеративную модель разработки, включающую четыре фазы: начало, исследование, построение и внедрение. Каждая фаза может быть разбита на этапы (итерации), в результате которых выпускается версия для внутреннего или внешнего использования. Прохождение через четыре основные фазы называется циклом разработки, каждый цикл завершается генерацией версии системы. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы. Суть работы в рамках *RUP* - это создание и сопровождение моделей на базе *UML [5]*.

– *Microsoft Solution Framework* (*MSF*) сходна с *RUP*, так же включает четыре фазы: анализ, проектирование, разработка, стабилизация, является итерационной, предполагает использование объектно-ориентированного моделирования. *MSF* в сравнении с *RUP* в большей степени ориентирована на разработку бизнес-приложений.

– *Extreme Programming* (*XP*). Экстремальное программирование (самая новая среди рассматриваемых методологий) сформировалось в 1996 году. В основе методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС, а разработка ведется с использованием последовательно дорабатываемых прототипов.

1.2 Развитие представлений о разработке программ: от спагетти-кода к методологии структурного программирования. Объектно-ориентированное программирование в Java.

Структу́рное программи́рование – парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. Концептуализирована в конце 1960-х — начале 1970-х годов на фундаменте теоремы Бёма — Якопини, математически обосновывающей возможность структурной организации программ, и работы Эдсгера Дейкстры «О вреде оператора goto» (англ. Goto considered harmful).

В соответствии с парадигмой, любая программа, которая строится без использования оператора goto, состоит из трёх базовых

управляющих конструкций: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения. В 1970-е годы объёмы и сложность программ достигли такого уровня, что традиционная (неструктурированная) разработка программ перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать. Поэтому потребовалась систематизация процесса разработки и структуры программ.

Методология структурной разработки программного обеспечения была признана «самой сильной формализацией 70-х годов».

По мнению Бертрана Мейера, «Революция во взглядах на программирование, начатая Дейкстрой, привела к движению, известному как структурное программирование, которое предложило систематический, рациональный подход к конструированию программ. Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование».

Структурное программирование призвано, в частности, устранить беспорядок и ошибки в программах, вызванные трудностями чтения кода, несистематизированным, неудобным для восприятия и анализа исходным текстом программы. Такой текст нередко характеризуют как «спагетти-код».

Спагетти-код — плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа, содержащая много операторов goto (особенно переходов назад), исключений и других конструкций, ухудшающих структурированность[8]. Один из самых известных антипаттернов программирования.

Спагетти-код назван так потому, что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный. Иногда называется «кенгуру-код» (kangaroo code) из-за множества инструкций jump.

В настоящее время термин применяется не только к случаям злоупотребления goto, но и к любому «многосвязному» коду, в

котором один и тот же небольшой фрагмент исполняется в большом количестве различных ситуаций и выполняет много различных логических функций [8].

Спагетти-код может быть отлажен и работать правильно и с высокой производительностью, но он крайне сложен в сопровождении и развитии [8]. Доработка спагетти-кода для добавления новой функциональности иногда несет значительный потенциал внесения новых ошибок. По этой причине становится практически неизбежным рефакторинг — главное лекарство от спагетти.

Принципы структурного программирования

Становление и развитие структурного программирования связано с именем Эдсгера Дейкстры.

Принцип 1. **Следует отказаться от использования оператора безусловного перехода goto.**

Принцип 2. **Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.**

- Последовательность — однократное выполнение операций в том порядке, в котором они записаны в тексте программы.

Бертран Мейер поясняет: «Последовательное соединение: используйте выход одного элемента как вход к другому, подобно тому, как электронщики соединяют выход сопротивления со входом конденсатора».

- Ветвление — однократное выполнение одной из двух или более операций, в зависимости от выполнения заданного условия.
- Цикл — многократное исполнение одной и той же операции до тех пор, пока выполняется заданное условие (условие продолжения цикла).

Принцип 3. **В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.** Никаких других средств управления последовательностью выполнения операций не предусматривается.

Принцип 4. **Повторяющиеся фрагменты программы можно оформить в виде** подпрограмм (процедур и функций). Таким же образом (в виде подпрограмм) можно оформить логически целостные фрагменты программы, даже если они не повторяются.

- В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется

инструкция «Вызов подпрограммы». При выполнении такой инструкции работает вызванная подпрограмма. После этого продолжается исполнение основной программы, начиная с инструкции, следующей за командой «Вызов подпрограммы».

- Бертран Мейер поясняет: «Преобразуйте элемент, возможно, с внутренними элементами, в подпрограмму, характеризуемую одним входом и одним выходом в потоке управления».

Принцип 5. **Каждую логически законченную группу инструкций следует оформить как блок.** Блоки являются основой структурного программирования.

Блок — это логически сгруппированная часть исходного кода, например, набор инструкций, записанных подряд в исходном коде программы. Понятие *блок* означает, что к блоку инструкций следует обращаться как к единой инструкции. Блоки служат для ограничения области видимости переменных и функций. Блоки могут быть пустыми или вложенными один в другой. Границы блока строго определены. Например, в if-инструкции блок ограничен кодом BEGIN..END (в языке Паскаль) или фигурными скобками {…} (в языке C) или отступами (в языке Питон).

Принцип 6. **Все перечисленные конструкции должны иметь один вход и один выход.**

Произвольные управляющие конструкции (такие, как в блюде спагетти) могут иметь произвольное число входов и выходов. Ограничив себя управляющими конструкциями с одним входом и одним выходом, мы получаем возможность построения произвольных алгоритмов любой сложности с помощью простых и надежных механизмов.

Принцип 7. **Разработка программы ведётся пошагово, методом «сверху вниз»** *(top-down method)*

Объе́ктно ориенти́рованное программи́рование (сокр. ООП) — методология или стиль программирования на основе описания типов/моделей предметной области и их взаимодействия, представленных порождением из прототипов или как экземпляры классов, которые образуют иерархию наследования.

Идеологически, ООП — подход к программированию как к моделированию информационных объектов, решающий на более высоком абстрактном уровне основную задачу структурного программирования — структурирование информации с точки зрения

управляемости. Это позволяет управлять самим процессом моделирования и реализовывать крупные программные проекты.

Управляемость для иерархических систем предполагает минимизацию избыточности данных (аналогичную нормализации) и их целостность, поэтому созданное удобно управляемым — будет и удобно пониматься. Таким образом, через тактическую задачу управляемости решается стратегическая задача — транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

- Абстракция для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;
- Инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;
- Наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;
- Полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

То есть фактически речь идёт о прогрессирующей организации информации согласно первичным семантическим критериям: «важное/неважное», «ключевое/подробности», «родительское/дочернее», «единое/множественное». Прогрессирование, в частности, на последнем этапе даёт возможность перехода на следующий уровень детализации, что замыкает общий процесс.

Обычный человеческий язык в целом отражает идеологию ООП, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в

переносном смысле, что в итоге развивает выражение представления через имя предмета до полноценного понятия-класса.

Объектно-ориентированное программирование в Java.

В Java встроен набор ключевых классов, содержащих основные абстракции реального мира, с которым придется иметь дело вашим программам. Основой популярности Java являются встроенные классы-абстракции, сделавшие его языком, действительно независимым от платформы. Библиотеки, подобные MFC/COM, OWL, VCL, nextstep, Motif и opendoc прекрасно работают на своих платформах, однако сегодня главной платформой становится Internet.

В реализации Java находится 23 пакета, а количество классов – 503

Таблица 1.1 Пакеты Java API

Имя пакета	Содержимое
Java.applet	Классы для реализации апплетов
Java.awt	Классы для работы с графикой, текстом, окнами и GUI
Java.awt.datatransfer	Классы для обеспечения передачи информации (Copy/Paste)
Java.awt.event	Классы и интерфейсы для обработки событий
Java.awt.image	Классы для обработки изображений
Java.awt.peer	GUI для обеспечения независимости от платформы
Java.beans	API для модели компонентов javabeans
Java.io	Классы для различных типов ввода-вывода
Java.lang	Классы ядра языка (типы, работа со строками, тригонометрические функции, обработка исключений, легковесные процессы)
Java.lang.reflect	Классы Reflection API
Java.math	Классы для арифметических операций произвольной точности
Java.net	Классы для работы в сети Интернет (сокеты, протоколы, URL)
Java.rmi	Классы, связанные с RMI (удаленный вызов процедур)
Java.rmi.dgc	Классы, связанные с RMI

Java.rmi.registry	Классы, связанные с RMI
Java.rmi.server	Классы, связанные с RMI
Java.security	Классы для обеспечения безопасности
Java.security.acl	Классы для обеспечения безопасности
Java.security.interfaces	Классы для обеспечения безопасности
Java.sql	
Java.text	Классы для обеспечения многоязыковой поддержки
Java.text.resources	Классы для обеспечения многоязыковой поддержки
Java.util	Разноообразные полезные типы данных (стеки, сдовари, хэш-таблицы, даты, генератор случайных чисел)
Java.util.zip	Классы для обеспечения архивации

Исходный файл на языке Java - это текстовый файл, содержащий в себе одно или несколько описаний классов. Транслятор Java предполагает, что исходный текст программ хранится в файлах с расширениями Java. Получаемый в процессе трансляции код для каждого класса записывается в отдельном выходном файле, с именем совпадающем с именем класса, и расширением class.

Прежде всего, в этой лекции мы напишем, оттранслируем, и запустим каноническую программу “Hello World”. После этого мы рассмотрим все существенные лексические элементы, воспринимаемые Java-транслятором: пробелы, комментарии, ключевые слова, идентификаторы, литералы, операторы и разделители. К концу главы вы получите достаточно информации для того чтобы самостоятельно ориентироваться в хорошей Java-программе.

Hello World

Итак, вот ваша первая Java-программа:

Class helloworld { public static void main (String args []) {
System. Out. Println ("Hello World");
} }

Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов. Приведенный выше текст примера надо записать в файл helloworld.java. Обязательно проверьте соответствие прописных букв в имени файла тому же в названии

содержащегося в нем класса. Для того, чтобы оттранслировать этот пример необходимо запустить транслятор Java — javac, указав в качестве параметра имя файла с исходным текстом:

C: \> javac helloworld.Java

Транслятор создаст файл helloworld.class с независимым от процессора байт-кодом нашего примера. Для того, чтобы исполнить полученный код, необходимо иметь среду времени выполнения языка Java (в нашем случае это программа java), в которую надо загрузить новый класс для исполнения.

Подчеркнем, что указывается имя класса, а не имя файла, в котором этот класс содержится.

C: > java helloworld

Hello World

Полезного сделано мало, однако мы убедились, что установленный Javaтранслятор и среда времени выполнения работают.

Шаг за шагом

Конечно, helloworld — это тривиальный пример. Однако даже такая простая программа новичку в языке Java может показаться пугающе сложной, поскольку она знакомит вас с массой новых понятий и деталей синтаксиса языка Давайте внимательно пройдемся по каждой строке нашего первого примера, анализируя те элементы, из которых состоит Java-программа. Строка 1 class helloworld {

В этой строке использовано зарезервированное слово class. Оно говорит транслятору, что мы собираемся описать новый класс. Полное описание класса располагается между открывающей фигурной скобкой в первой строке и парной ей закрывающей фигурной скобкой в строке 5. Фигурные скобки в Java используются точно так же, как в языках C и C++. Строка 2 public static void main (String args []) {

Такая, на первый взгляд, чрезмерно сложная строка примера является следствием важного требования, заложенного при разработке языка Java. Дело в том, что в Java отсутствуют глобальные функции. Поскольку подобные строки будут встречаться в большинстве примеров первой части книги, давайте пристальнее рассмотрим каждый элемент второй строки. Public

Разбивая эту строку на отдельные лексемы, мы сразу сталкиваемся с ключевым словом public. Это — *модификатор доступа,* который позволяет программисту управлять видимостью любого метода и любой переменной. В данном случае модификатор доступа public

означает, что метод main виден и доступен любому классу. Существуют еще 2 указателя уровня доступа — private и protected, с которыми мы более детально познакомимся в лекции 7. Static
Следующее ключевое слово — static. С помощью этого слова объявляются методы и переменные класса, используемые для работы с классом в целом. Методы, в объявлении которых использовано ключевое слово static, могут непосредственно работать только с локальными и статическими переменными. Void
У вас нередко будет возникать потребность в методах, которые возвращают значение того или иного типа: например, int для целых значений, float - для вещественных или имя класса для типов данных, определенных программистом. В нашем случае нужно просто вывести на экран строку, а возвращать значение из метода main не требуется. Именно поэтому и был использован модификатор void. Более детально этот вопрос обсуждается в лекции 3. Main
Наконец, мы добрались до имени метода main. Здесь нет ничего необычного, просто все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода main. Java-транслятор может оттранслировать класс, в котором нет метода main. А вот Java-интерпретатор запускать классы без метода main не умеет.
Все параметры, которые нужно передать методу, указываются внутри пары круглых скобок в виде списка элементов, разделенных символами ";" (точка с запятой). Каждый элемент списка параметров состоит из разделенных пробелом типа и идентификатора. Даже если у метода нет параметров, после его имени все равно нужно поставить пару круглых скобок. В примере, который мы сейчас обсуждаем, у метода main только один параметр, правда довольно сложного типа.
Элемент String args[] объявляет параметр с именем args, который является массивом объектов — представителей класса String. Обратите внимание на квадратные скобки, стоящие после идентификатора args. Они говорят о том, что мы имеем дело с массивом, а не с одиночным элементом указанного типа. Мы вернемся к обсуждению массивов в следующей лекции, а пока отметим, что тип String — это класс. Более детально о строках мы поговорим в лекции 8.
Строка 3
System. Out. Prlntln("Hello World!");

В этой строке выполняется метод println объекта out. Объект out объявлен в классе outputstream и статически инициализируется в классе System. В Лекциях 8 и 12 у вас будет шанс познакомиться с нюансами работы классов String и outputstream.
Закрывающей фигурной скобкой в строке 4 заканчивается объявление метода main, а такая же скобка в строке 5 завершает объявление класса helloworld.
Лексические основы
Теперь, когда мы подробно рассмотрели минимальный Java-класс, давайте вернемся назад и рассмотрим общие аспекты синтаксиса этого языка. Программы на Java — это набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей. Пробелы
Java — язык, который допускает произвольное форматирование текста программ. Для того, чтобы программа работала нормально, нет никакой необходимости выравнивать ее текст специальным образом. Например, класс helloworld можно было записать в двух строках или любым другим способом, который придется вам по душе. И он будет работать точно так же при условии, что между отдельными лексемами (между которыми нет операторов или разделителей) имеется по крайней мере по одному пробелу, символу табуляции или символу перевода строки.
Комментарии
Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:
A = 42; // если 42 - ответ, то каков же был вопрос?
Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст комментариев символами /* и закончив символами */ При этом весь текст между этими парами символов будет расценен как комментарий и транслятор его проигнорирует.
/*

** Этот код несколько замысловат...*

** Попробую объяснить:*

** .... */*

Третья, особая форма комментариев, предназначена для сервисной программы *javadoc,* которая использует компоненты Java-транслятора для автоматической генерации документации по интерфейсам классов. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением открытого (public) класса, метода или переменной документирующий комментарий, нужно начать его с символов /** (косая черта и две звездочки). Заканчивается такой комментарий точно так же, как и обычный комментарий — символами */. Программа javadoc умеет различать в документирующих комментариях некоторые специальные переменные, имена которых начинаются с символа @. Вот пример такого комментария:

*/***

** Этот класс умеет делать замечательные вещи. Советуем всякому, кто*

** Захочет написать еще более совершенный класс, взять его в качестве * базового.*

** @see Java. Applet. Applet*

** ©author Patrick Naughton*

** @version 1. 2*

/ class coolapplet extends Applet { / * У этого метода два параметра:*

** @param key - это имя параметра.*

** @param value - это значение параметра с именем key.*

**/ void put (String key, Object value) {* Зарезервированные ключевые слова Зарезервированные ключевые слова — это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. На сегодняшний день в языке Java имеется 59 зарезервированных слов (см. Таблицу 1.2). Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов.

Таблица 1.2 Зарезервированные слова Java

Abstract	Boolean	Break	Byte	Byvalue

Case	Cast	Catch	Char	Class
Const	Continue	Default	Do	Double
Else	Extends	False	Final	Finally
Float	For	Future	Generic	Goto
If	Implements	Import	Inner	Instanceof
Int	Interface	Long	Native	New
Null	Operator	Outer	Package	Private
Protected	Public	Rest	Return	Short
Static	Super	Switch	Syncronized	This
Throw	Throws	Transient	True	Try
Var	Void	Volatile	While	

Отметим, что слова byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var зарезервированы в Java, но пока не используются Кроме этого, в Java есть зарезервированные имена методов (эти методы наследуются каждым классом, их нельзя использовать, за исключением случаев явного переопределения методов класса Object).

Таблица 1.3 Зарезервированные имена методов Java

Clone	Equals	Finalize	Getclass	Hashcode
Notify	Notifyall	Tostring	Wait	

Идентификаторы

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами, которые будут описаны ниже. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы.

Литералы

Константы в Java задаются их литеральным представлением. Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода. Типы будут рассмотрены в лекции 3.

Целые литералы

Целые числа — это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 — это целый литерал. В данном примере приведены десятичные числа, то есть числа с основанием 10 — именно те, которые мы повседневно используем вне мира компьютеров. Кроме десятичных, в качестве целых литералов могут использоваться также числа с основанием 8 и 16 — восьмеричные и шестнадцатиричные. Java распознает восьмеричные числа по стоящему впереди нулю. Нормальные десятичные числа не могут начинаться с нуля, так что использование в программе внешне допустимого числа 09 приведет к сообщению об ошибке при трансляции, поскольку 9 не входит в диапазон 0.. 7, допустимый для знаков восьмеричного числа. Шестнадцатиричная константа различается по стоящим впереди символам нуль-х (0х или 0Х). Диапазон значений шестнадцатиричной цифры — 0.. 15, причем в качестве цифр для значений 10.. 15 используются буквы от А до F (или от а до f). С помощью шестнадцатиричных чисел вы можете в краткой и ясной форме представить значения, ориентированные на использование в компьютере, например, написав Oxffff вместо 65535.

Целые литералы являются значениями типа int, которое в Java хранится в 32-битовом слове. Если вам требуется значение, которое по модулю больше, чем приблизительно 2 миллиарда, необходимо воспользоваться константой типа long. При этом число будет храниться в 64-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву L, указав таким образом, что данное число относится к типу long. Например, Ox7fffffffffffffL или 92233720368547758О7L — это значение, наибольшее для числа типа long. Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо экспоненциальном форматах. В обычном формате число состоит из некоторого количества десятичных цифр, стоящей после них десятичной точки, и следующих за ней десятичных цифр дробной части. Например, 2.0, 3.14159 и .6667 — это допустимые значения чисел с плавающей точкой, записанных в стандартном формате. В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок

определяется положительным или отрицательным десятичным числом, следующим за символом E или e. Примеры чисел в экспоненциальном формате: 6.022e23, 314159E-05, 2e+100. В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа double. Если вам требуется константа типа float, справа к литералу надо приписать символ F или f. Если вы любитель избыточных определений — можете добавлять к литералам типа double символ D или d. Значения используемого по умолчанию типа double хранятся в 64-битовом слове, менее точные значения типа float — в 32-битовых.

Логические литералы

У логической переменной может быть лишь два значения — true (истина) и false (ложь). Логические значения true и false не преобразуются ни в какое числовое представление. Ключевое слово true в Java не равно 1, а false не равно 0. В Java эти значения могут присваиваться только переменным типа boolean либо использоваться в выражениях с логическими операторами. Символьные литералы

Символы в Java — это индексы в таблице символов UNICODE. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (' '). Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов: - 'a', 'z', '@'. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей.

Переменные

Переменная — это основной элемент хранения информации в Javaпрограмме. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса, доступная всем методам данного класса.

Локальные области действия объявляются с помощью фигурных скобок. Объявление переменной

Основная форма объявления переменной такова:

Тип идентификатор [= значение] [, идентификатор [= значение 7...];

Тип — это либо один из встроенных типов, то есть, byte, short, int, long, char, float, double, boolean, либо имя класса или интерфейса. Мы подробно обсудим все эти типы в следующей лекции. Ниже приведено несколько примеров объявления переменных различных типов. Обратите внимание на то, что некоторые примеры включают в себя инициализацию начального значения. Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

В приведенном ниже примере создаются три переменные, соответствующие сторонам прямоугольного треугольника, а затем *с* помощью теоремы Пифагора вычисляется длина гипотенузы, в данном случае числа 5, величины гипотенузы классического прямоугольного треугольника со сторонами 3-4-5.

```
Class Variables { public static void main (String args []) {
Double a = 3; double b = 4; double c; c = Math.sqrt (a* a + b* b);
System.out.println ("c = "+ c);
} }
```

Ваш первый шаг

Мы уже многого достигли: сначала написали небольшую программу на языке Java и подробно рассмотрели, из чего она состоит (блоки кода, комментарии). Мы познакомились со списком ключевых слов и операторов, чье назначение будет подробно объяснено в дальнейших Лекциях.

Типы

Простые типы

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирова-

Ния. В Java имеется восемь простых типов: — byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы:

1. Целые. К ним относятся типы byte, short, int и long. Эти типы предназначены для целых чисел со знаком.
2. Типы с плавающей точкой — float и double. Они служат для представления чисел, имеющих дробную часть.
3. Символьный тип char. Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.
4. Логический тип boolean. Это специальный тип, используемый для представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций. Целые числа

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые. Например, если значение переменной типа byte равно в шестнадцатиричном виде 0x80, то это — число -1. Byte

Тип byte — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

Byte b; byte c = 0x55;

Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

Short short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип, поскольку он определен, как тип, в котором старший байт стоит первым. *Short s; short t = Ox55aa;* int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647.

Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. В ближайшие годы этот тип будет прекрасно соответствовать машинным словам не только 32-битовых процессоров, но и 64-битовых с поддержкой быстрой конвейеризации для выполнения 32-битного кода в режиме совместимости. Всякий раз, когда в одном выражении фигурируют переменные типов byte, short, int и целые литералы, тип всего выражения перед завершением вычислений приводится к int. *Int i;*

Int j = 0x55aa0000; long

Тип long предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

Long m;

Long n = Ox55aa000055aa0000;

Не надо отождествлять *разрядность* целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало *поведению* типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности хранит переменные типа byte и short в виде 32-битовых значений, поскольку этот размер соответствует машинному слову большинства современных компьютеров (СМ – 8 бит, 8086 – 16 бит, 80386/486 – 32 бит, Pentium – 64 бит).

Ниже приведена таблица разрядностей и допустимых диапазонов для различных типов целых чисел.

Имя	Разрядность	Диапазон
Long	64	-9, 223, 372, 036, 854, 775, 808.. 9, 223, 372, 036, 854, 775, 807
Int	32	-2, 147, 483, 648.. 2, 147, 483, 647
Short	16	-32, 768.. 32, 767
Byte	8	-128.. 127

Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой — float и double и операторов для работы с ними. Характеристики этих типов приведены в таблице. Float

В переменных с обычной, или *одинарной точностью,* объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита.

Float f;

Float f2 = 3. 14F; // обратите внимание на F, т.к. По умолчанию все лите-

Ралы double double

В случае *двойной точности,* задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Все *трансцендентные* математические функции, такие, как sin, cos, sqrt,

возвращают результат типа double. *Double d; double pi = 3. 14159265358979323846;* Приведение типа

Приведение типов (type casting) — одно из неприятных свойств C++, тем не менее приведение типов сохранено и в языке Java. Иногда возникают ситуации, когда у вас есть величина какого-то определенного типа, а вам нужно ее присвоить переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменнойприемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа byte или short в переменную типа int. Это называется *расширением (widening)* или *повышением (promotion),* поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа int всегда достаточно для хранения чисел из диапазона, допустимого для типа byte, поэтому в подобных ситуациях оператора явного приведения типа не требуется. Обратное в большинстве случаев неверно, поэтому для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Эту процедуру иногда называют *сужением (narrowing),* поскольку вы явно сообщаете транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного вам типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа int) к типу приемника (переменной типа byte). Если бы при такой операции целое значение выходило за границы допустимого для типа byte диапазона, оно было бы уменьшено путем деления по модулю на допустимый для byte диапазон (результат деления по модулю на число — это остаток от деления на это число).

Int a = 100; byte b = (byte) a;

Автоматическое преобразование типов в выражениях

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.

Byte a = 40;

Byte b = 50; byte c = 100; int d = a b / c;*

Результат промежуточного выражения (a* b) вполне может выйти за диапазон допустимых для типа byte значений. Именно поэтому Java автоматически повышает тип каждой части выражения до типа int, так что для промежуточного результата (a* b) хватает места.
Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений транслятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе трансляции. В нем мы пытаемся записать значение 50* 2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение об ошибке от транслятора — ведь при занесении int в byte может произойти потеря точности.
Byte b = 50; b = b 2;*
^ Incompatible type for =. Explicit cast needed to convert int to byte. (Несовместимый тип для =. Необходимо явное преобразование int в byte) Исправленный текст :

Byte b = 50; b = (byte) (b 2);* что приводит к занесению в b правильного значения 100.
Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной — long, то и тип всего выражения тоже повышается до long. Не забывайте, что все целые литералы, в конце которых не стоит символ L (или 1), имеют тип int.
Если выражение содержит операнды типа float, то и тип всего выражения автоматически повышается до float. Если же хотя бы один из операндов имеет тип double, то тип всего выражения повышается до double. По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип double. Приведенная ниже программа показывает, как повышается тип каждой величины в выражении для достижения соответствия со вторым операндом каждого бинарного оператора.
Class Promote { public static void main (String args []) { byte b = 42; char c = 'a'; short s = 1024; int i = 50000; float f = 5.67f; double d =.1234;
Double result = (f b) + (i/ c) - (d* s);*
System. Out. Println ((f b)+ "+ "+ (i / c)+ " - " + (d* s));*

System. Out. Println ("result = "+ result);
} }

Подвыражение f* b — это число типа float, умноженное на число типа byte. Поэтому его тип автоматически повышается до float. Тип следующего подвыражения i / c (int, деленный на char) повышается до int. Аналогично этому тип подвыражения d* s (double, умноженный на short) повышается до double. На следующем шаге вычислений мы имеем дело с тремя промежуточными результатами типов float, int и double. Сначала при сложении первых двух тип int повышается до float и получается результат типа float. При вычитании из него значения типа double тип результата повышается до double. Окончательный результат всего выражения — значение типа double.

Операторы

Операторы в языке Java — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить операцию с некоторыми операндами. Некоторые операторы требуют одного операнда, их называют *унарными.* Одни операторы ставятся перед операндами и называются *префиксными,* другие — после, их называют *постфиксными* операторами. Большинство же операторов ставят между двумя операндами, такие операторы называются *инфиксными* бинарными операторами. Существует тернарный оператор, работающий с тремя операндами.

В Java имеется 44 встроенных оператора. Их можно разбить на 4 класса - *арифметические, битовые, операторы сравнения* и *логические.*

Арифметические операторы

Арифметические операторы используются для вычислений так же как в алгебре (см. Таблицу со сводкой арифметических операторов ниже). Допустимые операнды должны иметь числовые типы. Например, использовать эти операторы для работы с логическими типами нельзя, а для работы с типом char можно, поскольку в Java тип char — это подмножество типа int.

Оператор	Результат	Оператор	Результат
+	Сложение	+ =	Сложение с присваиванием
-	Вычитание (также унарный минус)	-=	Вычитание с присваиванием

*	Умножение	* =	Умножение с присваиванием
/	Деление	/=	Деление с присваиванием
%	Деление по модулю	%=	Деление по модулю с присваиванием
++	Инкремент	--	Декремент

Четыре арифметических действия

Ниже, в качестве примера, приведена простая программа, демонстрирующая использование операторов. Обратите внимание на то, что операторы работают как с целыми литералами, так и с переменными.

```
Class basicmath  {  public static void int a = 1 + 1; int  b = a *  3;
main(String args[]) {
Int c = b / 4; int d = b - a; int e = -d;
   System.out.println("a =     " +    A);
   System.out.println("b =     " +    B);
   System.out.println("c =     " +    C);
   System.out.println("d =     " +    D);
   System.out.println("e =     " +    E);
   } }
```

Исполнив эту программу, вы должны получить приведенный ниже результат:

C: \> java basicmath *a = 2 b = 6 c = 1 d = 4*
E = -4

Оператор деления по модулю

Оператор деления по модулю, или оператор mod, обозначается символом %. Этот оператор возвращает остаток от деления первого операнда на второй. В отличие от C++, функция mod в Java работает не только с целыми, но и с вещественными типами. Приведенная ниже программа иллюстрирует работу этого оператора.

```
Class Modulus {
Public static void main (String args []) { int x = 42;  double y = 42.3;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
```

} }

Выполнив эту программу, вы получите следующий результат:
C:\> Modulus *x mod 10 = 2 y mod 10 = 2.3*

Арифметические операторы присваивания
Для каждого из арифметических операторов есть форма, в которой одновременно с заданной операцией выполняется присваивание. Ниже приведен пример, который иллюстрирует использование подобной разновидности операторов.

Class opequals { public static void main(String args[]) { int a = 1; int b = 2;
*Int c = 3; a += 5; b *= 4; c += a * b; c %= 6;*
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
} }

А вот и результат, полученный при запуске этой программы:
C:> Java opequals
A = 6 b = 8
C = 3

Инкремент и декремент
В С существует 2 оператора, называемых операторами инкремента и декремента (++ и --) и являющихся сокращенным вариантом записи для сложения или вычитания из операнда единицы. Эти операторы уникальны в том плане, что могут использоваться как в префиксной, так и в постфиксной форме. Следующий пример иллюстрирует использование операторов инкремента и декремента.

Class incdec { public static void main(String args[]) { int a = 1; int b = 2; int c = ++b; int d = a++;
C++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);

} }

Результат выполнения данной программы будет таким:
C:\ java incdec *a = 2 b = 3 c = 4 d = 1*

Целочисленные битовые операторы

Для целых числовых типов данных — long, int, short, char и byte, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. В таблице приведена сводка таких операторов. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

Оператор	Результат	Оператор	Результат
~	Побитовое унарное отрицание (NOT)		
&	Побитовое И (AND)	&=	Побитовое И (AND) с при-Сваиванием
|	Побитовое ИЛИ (OR)	|=	Побитовое ИЛИ (OR) с при-Сваиванием
^	Побитовое исключающее ИЛИ (XOR)	^=	Побитовое исключающее ИЛИ (XOR) с присваиванием
>>	Сдвиг вправо	>> =	Сдвиг вправо с присваиванием
>>>	Сдвиг вправо с заполнением нулями	>>>=	Сдвиг вправо с заполнением нулями с присваиванием
<<	Сдвиг влево	<<=	Сдвиг влево с присваиванием

Приоритеты операторов

В Java действует определенный порядок, или приоритет, операций. В элементарной алгебре нас учили тому, что у умножения и деления более высокий приоритет, чем у сложения и вычитания. В программировании также приходится следить и за приоритетами операций. В таблице указаны в порядке убывания приоритеты всех операций языка Java.

Высший

( )	[ ]	.	
~	!		
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		

В первой строке таблицы приведены три необычных оператора, о которых мы пока не говорили. Круглые скобки () используются для явной установки приоритета. Как вы узнали из предыдущей главы, квадратные скобки [] используются для индексирования переменной-массива. Оператор . (точка) используется для выделения элементов из ссылки на объект — об этом мы поговорим в лекции 8. Все же остальные операторы уже обсуждались в этой лекции.

Классы

Базовым элементом объектно-ориентированного программирования в языке Java является класс. Напомним, что классы в Java не обязательно должны содержать метод main. Единственное назначение этого метода — указать интерпретатору Java, откуда надо начинать выполнение программы. Для того, чтобы создать класс, достаточно иметь исходный файл, в котором будет присутствовать ключевое слово class, и вслед за ним — допустимый идентификатор и пара фигурных скобок для его тела.

Class Point {
}

Как вы помните из главы 2, *класс* — это шаблон для создания объекта. Класс определяет структуру объекта и его *методы,* образующие функциональный интерфейс. В процессе выполнения Java-программы система использует определения классов для создания представителей классов. Представители являются реальными *объектами.* Термины «представитель», «экземпляр» и «объект» взаимозаменяемы. Ниже приведена общая форма определения класса.

Class имя_класса extends имя_суперкласса { type переменная1_объекта:
Type переменная2_объекта: type переменнаяп_объекта:

Type имяметода1(список_параметров) { тело метода;
}
Type имяметода2(список_параметров) { тело метода;
}
Type имя методам(список_параметров) { тело метода;
}
}

Ключевое слово extends указывает на то, что *«имя_класса»* — это подкласс класса *«имя_суперкласса».* Во лекции классовой иерархии Java стоит единственный ее встроенный класс — Object. Если вы хотите создать подкласс непосредственно этого класса, ключевое слово extends и следующее за ним имя суперкласса можно опустить — транслятор включит их в ваше определение автоматически. Примером может служить класс Point, приведенный ранее.

Переменные представителей (instance variables)
Данные инкапсулируются в класс путем объявления переменных между открывающей и закрывающей фигурными скобками, выделяющими в определении класса его тело. Эти переменные объявляются точно так же, как объявлялись локальные переменные в предыдущих примерах. Единственное отличие состоит в том, что их надо объявлять вне методов, в том числе вне метода main. Ниже приведен фрагмент кода, в котором объявлен класс Point с двумя переменными типа int. *Class Point { int x, y;*
}
В качестве типа для переменных объектов можно использовать как любой из простых типов, описанных в лекции 4, так и классовые типы. Скоро мы добавим к приведенному выше классу метод main, чтобы его можно было запустить из командной строки и создать несколько объектов.

Оператор new
Оператор new создает экземпляр указанного класса и возвращает ссылку на вновь созданный объект. Ниже приведен пример создания и присваивание переменной p экземпляра класса Point.
Point p = new Point();
Вы можете создать несколько ссылок на один и тот же объект. Приведенная ниже программа создает два различных объекта класса

Point и в каждый из них заносит свои собственные значения. Оператор точка используется для доступа к переменным и методам объекта.

```
Class twopoints {
Public static void main(String args[]) {
Point p1 = new Point(); Point p2 = new Point(); p1.x = 10; p1.y = 20;
p2.x = 42; p2.y = 99;
System.out.println("x = " + p1.x + " y = " + p1.y);
        System.out.println("x =          " + p2.x + " y = " + p2.y);
} }
```

В этом примере снова использовался класс Point, было создано два объекта этого класса, и их переменным x и y присвоены различные значения. Таким образом мы продемонстрировали, что переменные различных объектов независимы на самом деле. Ниже приведен результат, полученный при выполнении этой программы.

C:\> Java twopoints *x = 10 y = 20*
X = 42 y = 99

ЗАМЕЧАНИЕ

Поскольку при запуске интерпретатора мы указали в командной строке не класс Point, а класс twopoints, метод main класса Point был полностью проигнорирован. Добавим в класс Point метод main и, тем самым, получим законченную программу.

```
Class Point { int x, y; public static void main(String args[]) { Point p =
new Point(); p.x = 10;
P.y = 20;
        System.out.println("x =          " + p.x + " y = " + p.y);
} }
```

Объявление методов

Методы - это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Общая форма объявления метода такова:

Тип имя_метода (список формальных параметров) { тело метода:

}

Тип результата, который должен возвращать метод может быть любым, в том числе и типом void - в тех случаях, когда возвращать результат не требуется. Список формальных параметров - это последовательность пар типидентификатор, разделенных запятыми. Если у метода параметры отсутствуют, то после имени метода должны стоять пустые круглые скобки. *Class Point { int x, y; void init(int a, int b) {*
X = a; У = b;
} }

Вызов метода

В Java отсутствует возможность передачи параметров *по ссылке* на примитивный тип. В Java все параметры примитивных типов передаются *по значению,* а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке, можно изменять содержимое того объекта, на который ссылается данная переменная. В лекции 11 Вы узнаете, как предать переменные примитивных типов по ссылке (через обрамляющие классы-оболочки).

Скрытие переменных представителей

В языке Java не допускается использование в одной или во вложенных областях видимости двух локальных переменных с одинаковыми именами. Интересно отметить, что при этом не запрещается объявлять формальные параметры методов, чьи имена совпадают с именами переменных представителей. Давайте рассмотрим в качестве примера иную версию метода init, в которой формальным параметрам даны имена x и y, а для доступа к одноименным переменным текущего объекта используется ссылка this. *Class Point { int x, y; void init(int x, int y) { this.x = x; this.y = y } }*
class twopointsinit {
Public static void main(String args[]) {
Point p1 = new Point(); Point p2 = new Point();
P1.init(10,20); p2.init(42,99);
System.out.println("x = " + p1.x + " y = •• + p-l.y);
System.out.printlnc'x = " + p2.x + " y = •• + p2.y);
} }

Конструкторы

Инициализировать все переменные класса всякий раз, когда создается его очередной представитель — довольно утомительное дело даже в том случае, когда в классе имеются функции, подобные методу init. Для этого в Java предусмотрены специальные методы, называемые конструкторами. Конструктор — это метод класса, который инициализирует новый объект после его создания. Имя конструктора всегда *совпадает* с именем класса, в котором он расположен (также, как и в C++). У конструкторов нет типа возвращаемого результата - никакого, даже void. Заменим метод init из предыдущего примера конструктором.

```
Class Point { int x, y; Point(int x, int y) {  this.x = x; this.y = y;
} }
Class pointcreate {
Public static void main(String args[]) {
Point p = new Point(10,20);
System.out.println("x = " + p.x + " y = " + p.y);
} }
```

Программисты на Pascal (Delphi) для обозначения конструктора используют ключевое слово constructor.

Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещением методов (method overloading). В качестве примера приведена версия класса Point, в которой совмещение методов использовано для определения альтернативного конструктора, который инициализирует координаты x и y значениями по умолчанию (-1).

```
Class Point { int x, y; Point(int x, int y) {  this.x = x; this.y = y; } Point() {
x = -1; y = -1;
} }
Class pointcreatealt {
Public static void main(String args[]) {
Point p = new Point();
System.out.println("x = " + p.x + " y = " + p.y);
```

} }

В этом примере объект класса Point создается не при вызове первого конструктора, как это было раньше, а с помощью второго конструктора без параметров. Вот результат работы этой программы:
C:\> java pointcreatealt *x = -1 y = -1*

ЗАМЕЧАНИЕ
Решение о том, какой конструктор нужно вызвать в том или ином случае, принимается в соответствии с количеством и типом параметров, указанных в операторе new. Недопустимо объявлять в классе методы с одинаковыми именами и сигнатурами. В сигнатуре метода не учитываются имена формальных параметров учитываются лишь их типы и количество.

This в конструкторах
Очередной вариант класса Point показывает, как, используя this и совмещение методов, можно строить одни конструкторы на основе других. *Class Point { int x, y;*
Point(int x, int y) { this.x = x; this.y = y;
}
Point() { this(-1, -1); } }
В этом примере второй конструктор для завершения инициализации объекта обращается к первому конструктору.

Методы, использующие совмещение имен, не обязательно должны быть конструкторами. В следующем примере в класс Point добавлены два метода distance. Функция distance возвращает расстояние между двумя точками. Одному из совмещенных методов в качестве параметров передаются координаты точки x и y, другому же эта информация передается в виде параметраобъекта Point.

Class Point { int x, y; Point(int x, int y) { this.x = x; this. Y = y;
} double distance(int x, int y) { int dx = this.x - x; int dy = this.y - y;
*Return Math.sqrt(dx*dx + dy*dy);*
}
Double distance(Point p) {
Return distance(p.x, p.y);
} }

Class pointdist { public static void main(String args[]) {
Point p1 = new Point(0, 0);
Point p2 = new Point(30, 40);
System.out.println("p1 = " + pi.x + ", " + p1.y);
System.out.println("p2 = " + p2.x + ", " + p2.y);
System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(60, 80) = " + p1.distance(60, 80)); } }

Наследование и полиморфизм в java

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование (первый – инкапсуляция). Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и методы классов-предков. Классыпотомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс Point таким образом, чтобы включить в него третью координату z.
Class Point3D extends Point { int z; Point3D(int x, int y, int z) { this.x = x; this.y = y; this.z = z; } Point3D() { this(-1,-1,-1);
} }

В этом примере ключевое слово extends используется для того, чтобы сообщить транслятору о намерении создать подкласс класса Point. Как видите, в этом классе не понадобилось объявлять переменные x и y, поскольку Point3D унаследовал их от своего суперкласса Point.

Super
В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе. Вспомните, как во втором конструкторе мы использовали this для вызова первого конструктора того же класса. Аналогичным образом ключевое слово super позволяет обратиться непосредственно к конструктору суперкласса (в Delphi / C++ для этого используется ключевое слово inherited).

Class Point3D extends Point { int z; Point3D(int x, int y, int z) { super(x, y); // Здесь мы вызываем конструктор суперкласса this.z=z;
Public static void main(String args[]) {

```
Point3D p = new Point3D(10, 20, 30);
System.out.println( " x = " + p.x + " y = " + p.y +
                    " z = " + p.z);
} }
```

Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (пример pointdist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(mt x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны *заместить* (override) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и *совмещение* (overloading), и *замещение* (overriding) метода distance.

```
Class Point { int x, y; Point(int x, int y) { this.x = x; this.y = y;
}  double distance(int x, int y) { int dx = this.x - x; int dy = this.y - y:
Return Math,sqrt(dx*dx + dy*dy);
}
Double distance(Point p) { return distance(p.x, p.y);
}
}
        Class          Point3D extends Point { int z;
Point3D(int x, int y, int z)      { super(x, y);
This.z = z;
(
Double distance(int x, int y, int z) { int dx = this.x - x; int dy = this.y - y;
Int dz = this.z - z;
Return Math.sqrt(dx*dx + dy*dy + dz*dz);
}
Double distance(Point3D other) { return distance(other.x, other.y,
other.z);
}
Double distance(int x, int y) { double dx = (this.x / z) - x; double dy =
(this.y / z) - y;
Return Math.sqrt(dx*dx + dy*dy);
}
}
```

```
Class Point3DDist { public static void main(String args[]) { Point3D p1 =
new Point3D(30, 40, 10);
Point3D p2 = new Point3D(0, 0, 0);
Point p = new Point(4, 6);
        System.out.println("p1 = " + p1.x +        ", " + p1.y + ", " + p1.z);
        System.out.println("p2 = " + p2.x +        ", " + p2.y + ", " + p2.z);
System.out.println("p = " + p.x + ", " + p.y);
System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
System.out.println("p1.distance(p) = " + p1.distance(p));
} }
```

Ниже приводится результат работы этой программы:
C:\> Java Point3DDist *p1 = 30, 40, 10 p2 = 0, 0, 0 p = 4, 6*
P1.distance(p2) = 50.9902 p1.distance(4, 6) = 2.23607
P1.distance(p) = 2.23607

Обратите внимание — мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется *динамическим назначением методов* (dynamic method dispatch).

Динамическое назначение методов
Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс / суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
Class A { void callme() {
System.out.println("Inside A's callrne method"); class B extends A { void
callme() {
System.out.println("Inside B's callme method");
} }
Class Dispatch { public static void main(String args[]) {  A a = new B();
a.callme();
} }
```

Обратите внимание — внутри метода main мы объявили переменную а класса А, а проинициализировали ее ссылкой на объект класса В. В следующей строке мы вызвали метод callme. При этом транслятор

проверил наличие метода callme у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а callme класса B.

Final

Все методы и переменные объектов могут быть замещены по умолчанию. Если же вы хотите объявить, что подклассы не имеют права замещать какиелибо переменные и методы вашего класса, вам нужно объявить их как final (в Delphi / C++ не писать слово virtual).
Final int FILE_NEW = 1;
По общепринятому соглашению при выборе имен переменных типа final — используются только символы верхнего регистра (т.е. Используются как аналог препроцерных констант C++). Использование final-методов порой приводит к выигрышу в скорости выполнения кода — поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы *встроенным* (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

Finalize

В Java существует возможность объявлять методы с именем finalize. Методы finalize аналогичны деструкторам в C++ (ключевой знак ~) и Delphi (ключевое слово destructor). Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

Static

Иногда требуется создать метод, который можно было бы использовать вне контекста какого-либо объекта его класса. Так же, как в случае main, все, что требуется для создания такого метода — указать при его объявлении модификатор типа static. Статические методы могут непосредственно обращаться только к другим статическим методам, в них ни в каком виде не допускается использование ссылок this и super. Переменные также могут иметь тип static, они подобны глобальным переменным, то есть доступны из любого места кода. Внутри статических методов недопустимы ссылки на переменные представителей. Ниже приведен пример класса, у которого есть статические переменные, статический метод и статический блок инициализации.

```
Class Static { static int a = 3; static int b;
Static void method(int x) { System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}  static {
System.out.println("static block initialized"); b = a * 4;
}
Public static void main(String args[]) {
Method(42);
} }
```

Ниже приведен результат запуска этой программы.
C:\> java Static static block initialized
X = 42
A = 3
B = 12

В следующем примере мы создали класс со статическим методом и несколькими статическими переменными. Второй класс может вызывать статический метод по имени и ссылаться на статические переменные непосредственно через имя класса.

```
Class staticclass { static int a = 42; static int b = 99; static void callme() {
System.out.println("a = " + a);
} }
Class staticbyname { public static void main(String args[]) {
Staticclass.callme();
System.out.println("b = " + staticclass.b);
} }
```

А вот и результат запуска этой программы:
C:\> Java staticbyname *a = 42 b = 99*

Абстрактные классы и интерфейсы в Java

Бывают ситуации, когда нужно определить класс, в котором задана структура какой-либо абстракции, но полная реализация всех методов отсутствует. В таких случаях вы можете с помощью модификатора

типа abstract объявить, что некоторые из методов обязательно должны быть замещены в подклассах. Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

Abstract class A {
Abstract void callme(); void metoo() {
System.out.println("Inside A's metoo method");
} }
Class B extends A { void callme() {
System.out.println("Inside B's callme method");
} } class Abstract { public static void main(String args[]) { A a = new B(): a.callme():
A.metoo():
} }

В нашем примере для вызова реализованного в подклассе класса А метода callme и реализованного в классе А метода metoo используется динамическое назначение методов, которое мы обсуждали раньше.

C:\> Java Abstract
Inside B's callrne method Inside A's metoo method

Классическое заключение
В этой лекции вы научились создавать классы, конструкторы и методы. Вы осознали разницу между совмещением (overloading) и замещением (overriding) методов. Специальные переменные this и super помогут вам сослаться на текущий объект и на его суперкласс. В ходе эволюции языка Java стало ясно, что в язык нужно ввести еще несколько организационных механизмов - возможности более динамичного назначения методов и возможности более тонкого управления пространством имен класса и уровнями доступа к переменным и методам объектов. Оба этих механизма - *интерфейсы* и *пакеты,* описаны в следующей лекции.

Пакет (package) — это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем
List.
Интерфейс — это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию. Реализация же этих методов в интерфейсе отсутствует. Подобно абстрактным классам интерфейсы обладают замечательным дополнительным свойством — их можно многократно наследовать. Конкретный класс может быть наследником лишь одного суперкласса, но зато в нем может быть реализовано неограниченное число интерфейсов.

Пакеты
Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты — это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

Одиночный оператор package (необязателен) любое количество операторов import (необязательны) одиночное объявление открытого (public) класса любое количество закрытых (private) классов пакета (необязательны)

Оператор package
Первое, что может появиться в исходном файле Java — это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если

вы объявляете класс, как принадлежащий определенному пакету, например, *package java.awt.image;*
То и исходный код этого класса должен храниться в каталоге java/awt/image.

Ограничение доступа
Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов :

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.
- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: private (закрытый), public (открытый) и protected (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

	Private	Модифи-Катор отсутствует	Private protected	Protected	Public
Тот же класс	Да	Да	Да	Да	Да
Подкласс в том же пакете	Нет	Да	Да	Да	Да
Независимый класс в том же пакете	Нет	Да	Нет	Да	Да
Подкласс в другом пакете	Нет	Нет	Да	Да	Да
Независимый класс в другом пакете	Нет	Нет	Нет	Нет	Да

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет — используйте комбинацию private protected.

Ниже приведен довольно длинный пример, в котором представлены все допустимые комбинации модификаторов уровня доступа. В исходном коде первого пакета определяется три класса: Protection, Derived и samepackage. В первом из этих классов определено пять целых переменных — по одной на каждую из возможных комбинаций уровня доступа. Переменной n приписан уровень доступа по умолчанию, n_pri — уровень private, n_pro — protected, n_pripro — private protected и n_pub — public. Во всех остальных классах мы пытаемся использовать переменные первого класса. Те строки кода, которые из-за ограничения доступа привели бы к ошибкам при трансляции, закомментированы с помощью однострочных комментариев (//) — перед каждой указано, откуда доступ при такой комбинации модификаторов был бы возможен. Второй класс — Derived — является подклассом класса Protection и расположен в том же пакете p1. Поэтому ему доступны все перечисленные переменные за исключением n_pri. Третий класс, samepackage, расположен в том же пакете, но при этом не является подклассом Protection. По этой причине для него недоступна не только переменная n_pri, но и n_pripro, уровень доступа которой — private protected.

Package p1;
Public class Protection { int n = 1; private int n_pri = 2; protected int n_pro = 3; private protected int n_pripro = 4; public int n_pub = 5; public Protection() {
System.out.println("base constructor"); System.out.println("n = " + n);

```
System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = "       + n_pro);
System.out.println("n_pripro = " + n_pripro);
System.out.println("n_pub = " + n_pub);
} }
Class Derived extends Protection {
Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
        // только в классе
        // System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = "       + n_pro);
        System.out.println("n_pripro = " + n_pripro);
                System.out.println("n_pub = "       + n_pub);
} }
        Class         samepackage {
Samepackage() {
            Protection p = new Protection();
            System.out.println("same package constructor");
            System.out.println("n = " + p.n);
            // только в классе
            // System.out.println("n_pri = " + p.n_pri);
            System.out.println("n_pro = " + p.n_pro);
            // только в классе и подклассе
            //      System.out.println("n_pripro      =      "      +      p.n_pripro):
System.out.println("n_pub = " + p.n_pub):
} }
```

Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает

возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже: *interface имя { тип_результата имя_метода1(список параметров);*
Тип имя_final1-переменной = значение;
}
Обратите внимание — у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются final - переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса, содержащего единственный метод с именем callback и одним параметром типа int.
Interface Callback {
Void callback(int param);
}

Оператор implements

Оператор implements — это дополнение к определению класса, реализующего некоторый интерфейс(ы).
Class имя_класса [extends суперкласс]
[implements интерфейс0 [, интерфейс1...]] { тело класса }
Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми. Ниже приведен пример класса, в котором реализуется определенный нами интерфейс:

Class Client implements Callback {
Void callback(int p) {
System.out.println("callback called with " + p);

} }

В очередном примере метод callback интерфейса, определенного ранее, вызывается через переменную - ссылку на интерфейс:

Class testiface { public static void main(String args[]) { Callback c = new client(); c.callback(42);
} }

Ниже приведен результат работы программы:
C:\> Java testiface
Callback called with 42

Переменные в интерфейсах
Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы. Это аналогично использованию файловзаголовков для задания в C и C++ констант с помощью директив #define или ключевого слова const в Pascal / Delphi.

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором final. В приведенном ниже примере проиллюстрировано использование интерфейса для совместно используемых констант.

Import java.util.Random; interface sharedconstants { int NO = 0; int YES = 1; int MAYBE = 2; int LATER = 3; int SOON = 4; int NEVER = 5; } class Question implements sharedconstants {
Random rand = new Random();
*Int ask() { int prob = (int) (100 * rand.nextdouble()); if (prob < 30) return NO; // 30% else if (prob < 60) return YES; // 30% else if (prob < 75) return LATER; // 15% else if (prob < 98) return SOON; // 13% else return NEVER; // 2% } } class askme implements sharedconstants {*
Static void answer(int result) { switch(result) { case NO:
System.out.println("No"); break; case YES:

```
            System.out.println("Yes");
        Break; case MAYBE:
            System.out.println("Maybe");
        Break; case LATER:
            System.out.println("Later");
        Break; case SOON:
            System.out.priniln("Soon");
        Break; case NEVER:
            System.out.println("Never");
            Break;
} }
Public static void main(String args[]) {
Question q = new Question();        answer(q.ask());
        answer(q.ask());          answer(q.asko);
            Answer(q.ask());
} }
```

Обратите внимание на то, что результаты при разных запусках программы отличаются, поскольку в ней используется класс генерации случайных чисел Random пакета java.util. Описание этого пакета приведено в лекции 11.

```
C:\> Java askme
Later
Scon
No
Yes
```

Использование пакетов

Теперь вы обладаете полной информацией для создания собственных пакетов классов. Легко понимаемые интерфейсы позволят другим программистам использовать ваш код для самых различных целей. Инструменты, которые вы приобрели, изучив эту и предыдущую главы, должны вам помочь при разработке любых объектно-ориентированных приложений. В дальнейшем вы познакомитесь с некоторыми важными специфическими свойствами Java, которые представлены в виде классов в пакете java.lang.

2. ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПРОЕКТНЫХ РЕШЕНИЙ: ОБЩИЕ ПРИНЦИПЫ, МЕТОДЫ, СТАНДАРТЫ

Проектирование является важнейшим процессом при проектировании программного обеспечения (ПО). По этой причине, разработчики CASE-средств в своих продуктах вынуждены уделять моделированию данных повышенное внимание. Являясь признанным лидером в области объектных методологий, фирма Rational Software Corporation, тем не менее, до недавнего времени такого средства не имела. Основной причиной этого, повидимому, является ориентация на язык Unified Modeling Language (UML), как универсальный инструмент моделирования. UML полностью покрывает потребности моделирования данных. Сложившаяся на протяжении десятилетий технология моделирования данных, традиции, система понятий и колоссальный опыт разработчиков не могли далее игнорироваться. Немаловажную роль здесь сыграла и необходимость формального контроля моделей данных, что является абсолютно необходимым при проектировании мало-мальски больших схем баз данных и что UML не обеспечивает в достаточной степени. И, наконец, последней причиной, побудившей специалистов Rational Software Corporation к созданию собственного средства моделирования данных, является требование построения эффективных физических моделей, прежде всего для конкретных СУБД - лидеров рынка.

В начале 2000 года фирма Rational Software Corporation анонсировала появление собственного средства моделирования данных – Data Modeler, и в настоящее время оно доступно специалистам, например, использующим в своей работе Rational Rose 2000.

Целью данной лабораторной работы является знакомство с основными возможностями этого нового средства.

Авторы Data Modeler, прежде всего, ориентировались на создание инструмента проектирования физической модели данных. При этом не произошло отказа от UML как от средства моделирования данных, а некоторым образом были смещены акценты: теперь UML предполагается использовать для построения логической модели. По сути, логическая модель - это та же объектная модель, состоящая из объектов - сущностей. Переход от логической модели к физической и,

наоборот, в части моделирования данных обеспечивается Rational Rose автоматически. Для этого введено соответствие элементов моделей (табл. 2.1).

Таблица 2.1. Соответствие элементов логической и физической модели

Логическая модель	Физическая модель
Class (Класс)	Table (Таблица)
Operation (Операция)	Constraint (Ограничение)
Attribute (Атрибут)	Column (Колонка)
Package (Пакет)	Scheme (Схема)
Component (Компонент)	Database (База данных)
Association (Ассоциация)	Relationship (Связь)
Нет	Trigger (Тригер)
Нет	Index (Индекс)

Rose Data Modeler

После установки Rational Rose в специальной редакции (Rational Rose Professional Data Modeler Edition) в разделе главного меню Tools появляется новый раздел Data Modeler (рис. 2.1).

В разделе Data Modeler имеются два пункта: “Add Schema” и “Reverse Engeneer…”. Пункт “Add Schema” используется для создания новых схем БД, а пункт “Reverse Engeneer” - для построения модели на основе существующей схемы БД.

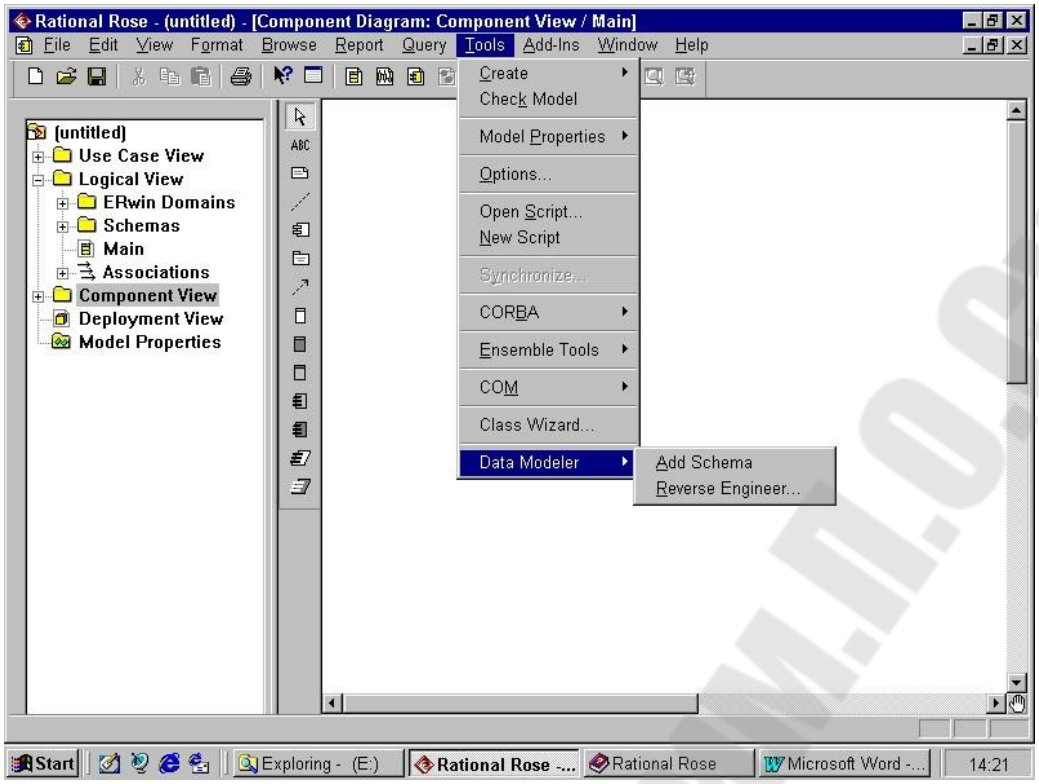

Рисунок 2.1- Отображение компоненты Data Modeler в меню Rational Rose Создание диаграммы модели данных

После создания схемы (рис. 2.1) в ней можно сформировать диаграмму модели данных. Эта диаграмма позволит добавить, изменить или просмотреть таблицы и другие элементы модели данных, поскольку играет ту же роль, что и диаграмма классов в объектной модели. Хотя можно добавлять элементы моделирования данных непосредственно в браузере, диаграмма модели данных позволяет показать в графическом виде, как сами элементы, так и отношения между ними. Для любой схемы можно создать любое необходимое число Создание диаграммы модели данных осуществляется следующим образом:

1. В браузере щелкните правой кнопкой мыши на схеме.
2. Выберите Data Modeler > New > Data Model Diagram.
3. Введите имя новой диаграммы.
4. Дважды щелкните на диаграмме для ее открытия.

Как и другие диаграммы в среде Rose, диаграмма модели данных имеет специализированную панель инструментов для добавления таблиц, отношений и других элементов моделирования. Кнопки этой панели перечислены в таблице 2.2.

Таблица 2.2 – Значки панели инструментов для диаграммы модели данных

Значки	Назначение
	Курсор принимает форму стрелки для выделения элемента Добавляет в диаграмму текстовое поле Добавляет к элементу диаграммы примечание Соединяет примечание с элементом диаграммы Добавляет в диаграмму таблицу Рисует неидентифицируемое отношение между двумя таблицами Рисует идентифицируемое отношение между двумя таблицами Добавляет в диаграмму представление Рисует зависимость между двумя таблицами

Диаграмма Data Model предоставляет следующие возможности:

- Создание и редактирование таблиц и их элементов (колонок, ограничений, индексов, триггеров и т. П.);
- Создание и редактирование идентифицирующих связей между таблицами;
- Создание и редактирование неидентифицирующих связей.

Основные возможности по работе с таблицей доступны, если войти в контекстном меню в пункт “Open Specification”. Появляющееся на экране окно включает следующую информацию (рис. 2.2).

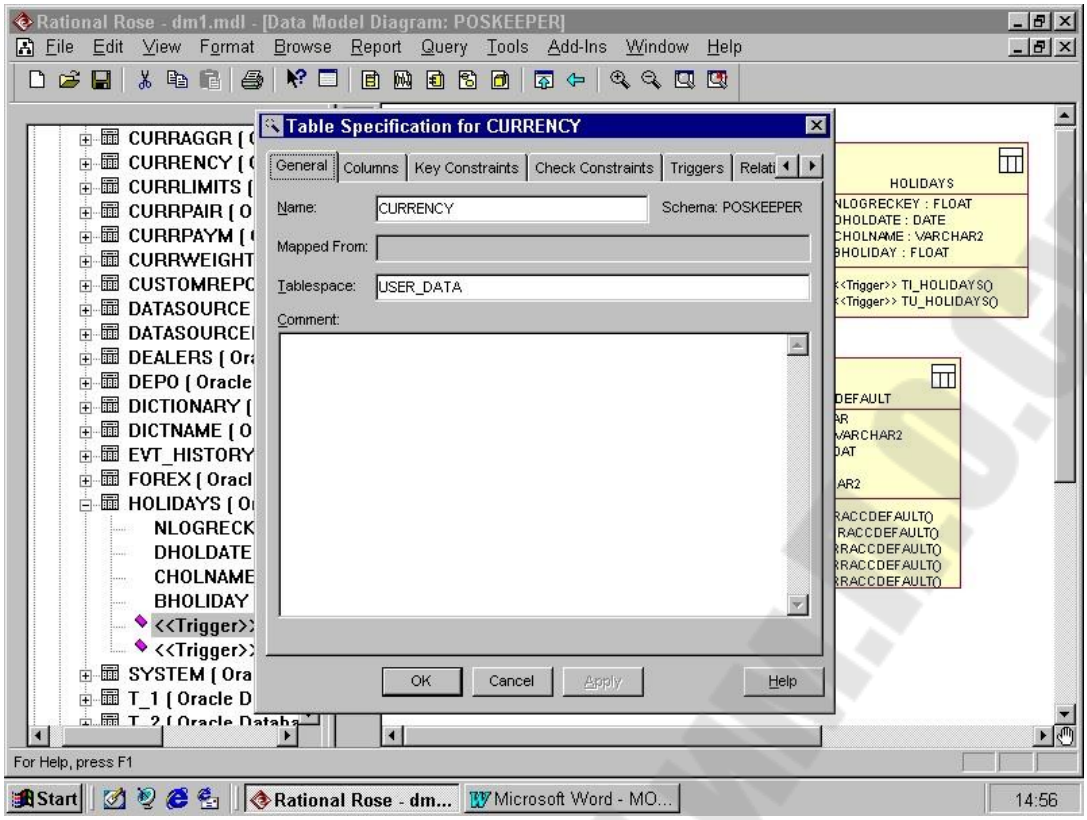

Рисунок 2.2 - Окно спецификации таблицы

При редактировании спецификации таблицы обеспечиваются следующие возможности (табл. 13.3).

Таблица 2.3 - Спецификация таблицы БД

Закладка	Описание
General	Вводится общая информация о таблице.
Columns	Задается описание колонок. Здесь можно добавить или отредактировать свойства колонок, задать тип, длину, обязательность (NULL, NOT NULL), а также пометить, что колонка входит в состав первичного ключа. Типы колонок соответствуют типам конкретной выбранной СУБД.
Key Constraints	Задаются ограничения на колонки таблицы. Здесь можно задать ограничение на уникальность первичного ключа, ограничение на уникальность альтернативных ключей, а также про-
	Сто определить индекс.

Check Constraints	Задаются выражения – инварианты, которые должны выполнятся для всех строк таблицы.
Triggers	Содержит список триггеров, который можно отредактировать, в том числе добавив новый триггер.
Relationships	При наличии связей между таблицами, закладка содержит полный список связей.

Добавление отношений

Отношения в модели данных подобны отношениям в объектной модели. В объектной модели отношение связывает два класса, а в модели данных — две таблицы. В Rose поддерживаются два основных типа отношений: идентифицируемые отношения (identifying relationship) и неидентифицируемые отношения (non-identifying relationship).

В обоих случаях для поддержки отношений в дочернюю таблицу добавляется внешний ключ. При идентифицируемом отношении внешний ключ становится частью первичного ключа в дочерней таблице. В этом случае дочерняя таблица не может содержать запись, не связанную с записью в родительской таблице. Идентифицируемые отношения моделируются составными агрегациями.

Неидентифицируемые отношения тоже создают внешний ключ в дочерней таблице, но он не становится частью первичного ключа в дочерней таблице. При неидентифицируемом отношении мощность (множественность) определяет то, будет ли запись в дочерней таблице существовать без связи с записью в родительской таблице. Если мощность равна 1, должна присутствовать родительская запись. Если мощность равна 0..1, присутствие родительской записи необязательно. Неидентифицируемые отношения моделируются ассоциациями.

Для редактирования свойств связи требуется войти в пункт контекстного меню “Open specification”.

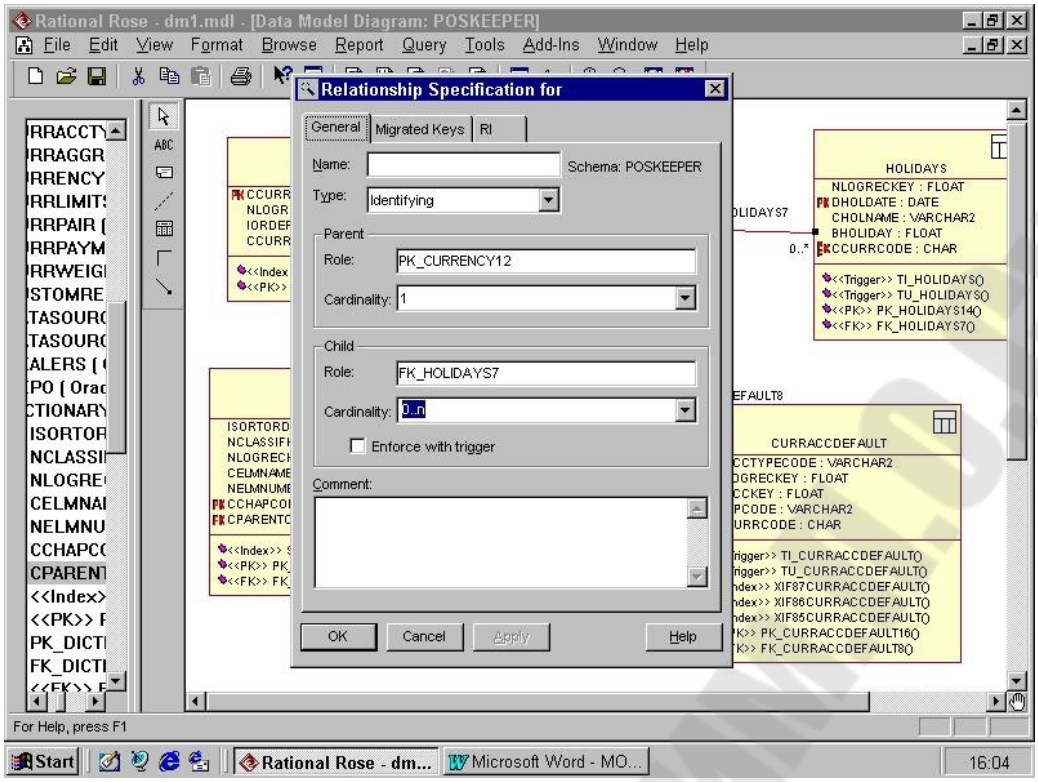

Рисунок 2.3 - Окно спецификации связи

При редактировании спецификации связи обеспечиваются следующие возможности (табл. 2.4).

Таблица 2.4 - Спецификация связи

Закладка	Описание
General	Основные свойства связи. Здесь задаются: Имя связи; тип связи; Наименования ролей (Parent, Child); кардинальность для каждой роли;
Migrated Key	Содержит список внешних ключей, образующихся в результате создания связи.

RI	Задание условий ссылочной целостности. Ссылочная целостность обеспечивается двумя способами: на основе триггеров; на основе декларативной ссылочной целостности (с использованием ограничений внешних ключей). Оба способа реализуют наиболее популярные алгоритмы, задаваемые для каждой роли (только для операций update и delete, для insert мы не нашли): Restrict; Cascade; Set Null; Set Default.

2.2 МОДЕЛИРОВАНИЕ И АЛГОРИТМИЗАЦИЯ КАК СРЕДСТВА ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

На рисунке 2.4 приведен фрагмент модели данных заданной предметной области. Данная модель разработана в среде Rational Rose 2003 с использованием утилиты Rose Data Modeler.

Описание предметной области.

Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается следующая информация:

- Предприятие, предоставляющее соответствующую вакансию;
- Название вакансии (должность);
- Требования к соискателю: пол, возраст, образование, знание определенных видов деятельности (выбор из перечня - знание электронного документооборота, определенных прикладных программ и т.п.), коммуникабельность;
- Обязанности (выбор из перечня – заключение договоров, распространение агитационного материала, работа с клиентами и т.п.);
- Предполагаемая оплата, единицы измерения оплаты - рубли;
- Оформление трудовой книжки (да, нет);
- Наличие социального пакета (да, нет);
- Срок начала открытия вакансии;

- Срок закрытия вакансии (вакансия занята).

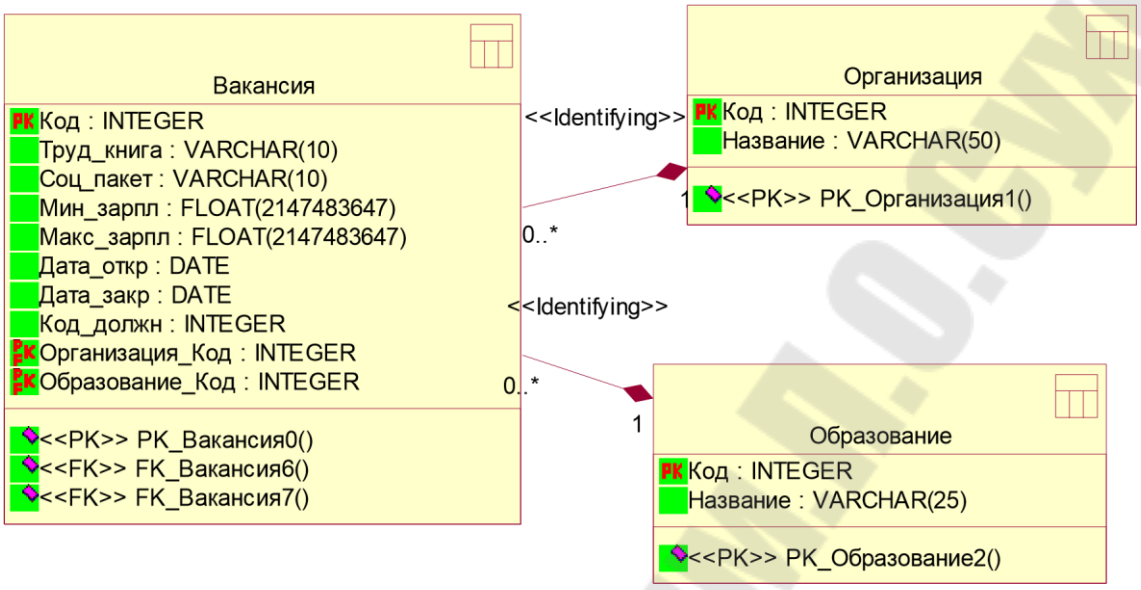

Рисунок 2.4 – Модель данных предметной области

Разработка диаграммы вариантов использования

Диаграмма вариантов использования отображает взаимодействие между вариантами использования, представляющими функции системы, и действующими лицами, представляющими людей или системы, получающие или передающие информацию в данную систему.

Данный тип диаграмм предназначен для создания списка операций, которые выполняет система, поэтому его иногда называют диаграммой функций. Любая система обладает своим множеством вариантов использования и множеством действующих лиц. Каждый вариант использования описывает элемент представляемой системой функциональности. Множество вариантов использования описывает всю функциональность системы на некотором уровне абстракции. Абстракция (abstraction) – сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Использование абстракции позволяет сохранить свободу принятия решений как можно дольше благодаря тому, что детали не фиксируются раньше времени. Каждое действующее лицо представляет собой один вид объектов, для которых система может выполнять некоторое поведение.

Язык UML предусматривает систему графических обозначений для вариантов использования (рис. 2.5).

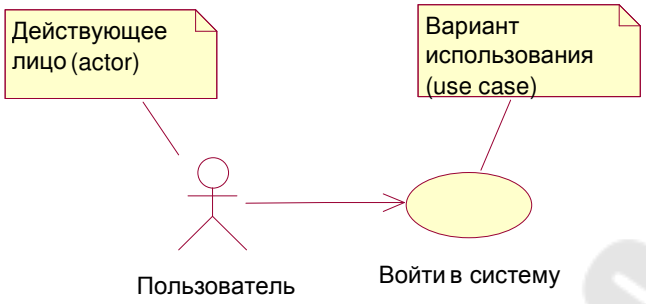

Рисунок 2.5 – Графические обозначения диаграммы вариантов использования

Действующее лицо (actor) – это непосредственный внешний пользователь системы. Это объект или множество объектов, непосредственно взаимодействующих с системой. Каждое действующее лицо является обобщением группы объектов, ведущих себя определенным образом по отношению к системе. Действующими лицами могут быть люди, устройства и другие системы – все, что взаимодействует с интересующей нас системой непосредственно.

Действующее лицо должно иметь одну четко определенную цель. Моделирование действующих лиц помогает определить границы системы, то есть идентифицировать объекты, находящиеся внутри системы, и объекты, лежащие на ее границе.

Различные взаимодействия действующих лиц с системой группируются в варианты использования. Вариант использования (use case) – это связный элемент функциональности, представляемый системой при взаимодействии с действующими лицами (рис. 2.6) .

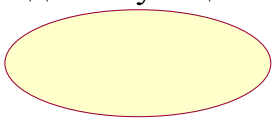

Рисунок 2.6 – Вариант использования

В каждом варианте использования участвуют одно или несколько действующих лиц и система.

Вариант использования объединяет все поведение, имеющее отношение к элементу функциональности системы: нормальное поведение, вариации нормального поведения, исключительные ситуации, сбойные ситуации и отмены запросов.

Любой вариант использования должен иметь краткое описание, объясняющее действия в этом варианте. Описание должно быть кратким, но в него необходимо включить сведения о разных типах пользователей, выполняющих данный вариант использования, и ожидаемый результат. Во время работы (особенно если проект сложный) эти описания будут напоминать членам команды, почему тот или иной вариант использования был включен в проект и что он должен делать. Четко документируя, таким образом, цели каждого варианта использования, можно уменьшить неразбериху, возникающую среди разработчиков.

На рисунке 2.7 приведен пример документирования варианта использования.

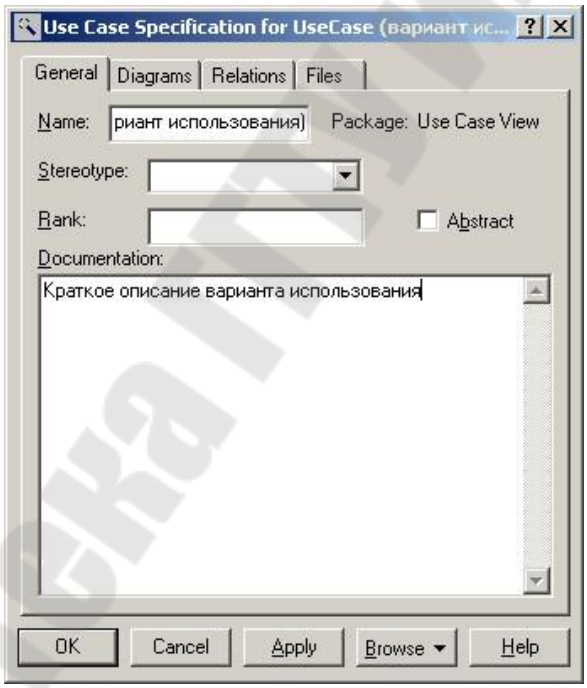

Рисунок 2.7 – Окно документирования варианта использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы.

В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис. Следует заметить, что два варианта использования, определенные для одной и той же сущности, не могут взаимодействовать друг с другом, поскольку каждый из них самостоятельно описывает законченный вариант использования этой сущности. Более того, варианты использования всегда предусматривают некоторые сигналы или сообщения, когда взаимодействуют с актерами за пределами системы. В то же время могут быть определены другие способы для взаимодействия с элементами внутри системы. В языке UML имеется несколько стандартных видов отношений между действующими лицами и вариантами использования:

- Отношение ассоциации;
- Отношение зависимости;
- отношение обобщения.

Ассоциация – структурное отношение, описывающее совокупность связей, представленных соединениями между объектами модели. Выделяют разновидность ассоциации, агрегирование, предусмотренное для выражения отношений между целым и его частями.

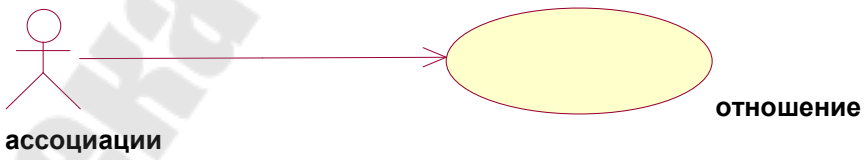

Рисунок 2.8 – Пример реализации отношения ассоциации

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это

отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа "*" (звездочка).

Зависимость – семантическое отношение между двумя сущностями, при которой изменение одной из сущностей, независимой, может повлиять на семантику второй сущности, зависимой.

Отношение зависимости определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение зависимости является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования.

Отношение зависимости между вариантами использования обозначается пунктирной линией со стрелкой, направленной от того варианта использования, который является расширением для исходного варианта использования.

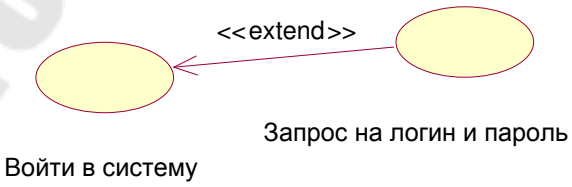

Рисунок 2.8 - Пример графического изображения отношения зависимости
(расширения) между вариантами использования

Обобщение – отношение, при котором объект специализированного элемента может быть подставлен и использован вместо объекта обобщенного элемента. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 2.9).

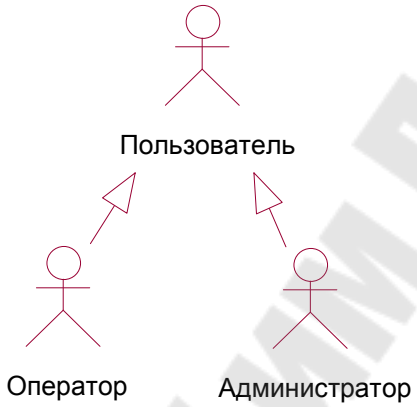

Рисунок 2.9 - Пример графического изображения отношения обобщения между действующими лицами

Особенности разработки диаграмм вариантов использования в среде IBM Rational Rose 2003

Запустите *Rational Rose*, и создайте новую пустую модель. Для этого в окне Create New Model мастера создания моделей, открывающегося после первого запуска системы, включите флажок Don't show this dialog in the future и нажмите кнопку *Cancel*. Эта команда закроет окно мастера, и не будет выводить его при следующих открытиях *Rational Rose*. В рабочем поле *Rational Rose* будет выведена пустое окно диаграммы классов. Это будет наша рабочая модель, в которой и должны быть отражены все нюансы будущей системы. Затем для того чтобы открыть окно диаграммы *Use Case* необходимо сделать двойной щелчок по значку *Main* в папке *User Case View* в окне Browser. Если открыто окно хотя бы одной диаграммы, то в главном меню активизируется пункт Browse и диаграмму *Use Case* можно открыть командой Browse,Use Case Diagram. Отметим, что в панели

инструментов Standard нет кнопки просмотра *Browse Use Case Diagram.*Создание новых элементов в диаграмме *Use Case*

Rational *Rose* предоставляет несколько способов создания новых элементов в окне диаграмм *Use Case*:

1. Командой New,Use Case контекстного меню папки Use Case View в окне Browser .
2. Командой Tools,Create,Use Case главного меню.
3. Командами строки инструментов окна Use Case Diagram.

В первом случае элемент создается непосредственно в дереве модели (в папке Use Case View окна Browser), но его значок не включается ни в одну диаграмму. После создания элемента, таким способом, можно поместить его на выбранную диаграмму, например путём перетаскивания мышкой значка элемента из дерева модели окна Browser в окно диаграммы. Во втором и третьем случае вместе с созданием элемента его значок помещается на текущую диаграмму автоматически.

При создании элементов посредством меню Tools программа предоставляет возможность создавать все элементы, которые можно включить в текущую диаграмму, тогда как при создании средствами строки инструментов пользователь ограничен созданием элементов согласно включенным в данную строку значкам. По причине большей простоты и наглядности рекомендуем пользоваться третьим вариантом. Для этого необходимо ознакомиться с содержанием строки инструментов, установленной по умолчанию для данной диаграммы. Для моделирования бизнес-процессов *Rational Rose* предоставляет дополнительные элементы *Use Case*, которые можно активизировать при помощи режима настройки инструментов (командой Use Case Diagram… на вкладке *Toolbars* окна Optins, открываемого командой Tools,Options главного меню). Но для создания системы учета товародвижения на складе достаточно значков, установленных в панелях инструментов по умолчанию.

Рассмотрим панель инструментов рабочего окна диаграммы Use Case.

Таблица 2.5 - Пиктограммы панели инструментов диаграммы Use Case

Пиктограмма	Кнопка	Описание
	Selects or deselects an item (Выделение или отмена	Превращает курсор в стрелку указа теля, так что вы можете выделить Объект

	выделения объекта)	
	Text Box (Текст)	Добавляет к диаграмме текст
	Note (Примечание)	Добавляет к диаграмме примечание
	Anchor Note to Item (Прикрепление примечания к объекту)	Связывает примечание с вариантом использования или объектом на диа-Грамме
	Package (Пакет)	Помещает на диаграмму новый пакет
	Use Case (Вариант использования)	Помещает на диаграмму новый вариант использования
	Actor (Действующее лицо	Помещает на диаграмму новое действующее лицо
	Unidirectional Association (Однонаправленная ассоциация)	Рисует связь между действующим лицом и вариантом использования
	Dependency or Instantiates (Зависимость или наполнение)	Рисует зависимость между элементами диаграммы
	Generalization (Обобщение)	Рисует связь использования или расширения между вариантами использования либо рисует связь наследования между действующими лицами

Способы именования элементов и связей

Структурным элементам Actor, Use Case и Package, вводимым в диаграмму, система *Rational Rose* автоматически присваивает умалчиваемые имена (*newclassn, newusecasen* и *newpackagen*). Изменит эти имена можно двумя способами:

– Редактирования подписи под значком после двойного щелчка на ней левой клавишей мыши (эта команда переводит окно с надписью в режим редактирования текста);

– Контекстной командой Rename на соответствующем значке в дереве модели окна Browser;
– Активизируя панель спецификаций элемента (двойным левым щелчком мыши на самом значке элемента в окне диаграммы) и редактируя умалчиваемое имя в поле *Name:* этой панели.

Связям умалчиваемые имена не присваиваются. При необходимости их именования можно использовать только третий способ – вводом нужного имени в поле *Name:* окна спецификаций связей. Для открытия этого окна переведите указатель мыши точно на стрелку связи в окне диаграммы и выполните двойной левый щелчок. Должно открыться диалоговое окно Assotiasin Specification for… с полем *Name:*, в которое и вводится имя связи.

Пример разработанной диаграммы вариантов использования

На рисунке 13.7 приведена диаграмма вариантов использования для приложения АИС «Трудоустройство».

Описание предметной области. Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается определенная информация. Помимо проектирования реляционной базы данных АИС, необходимо еще и разработать приложение к ней.

Цель приложения: автоматизация информационного процесса определения подходящей вакансии по данным резюме (анкеты) соискателя.

Метод: дискриминантный анализ.

Пользователь: менеджер по работе с кадрами.

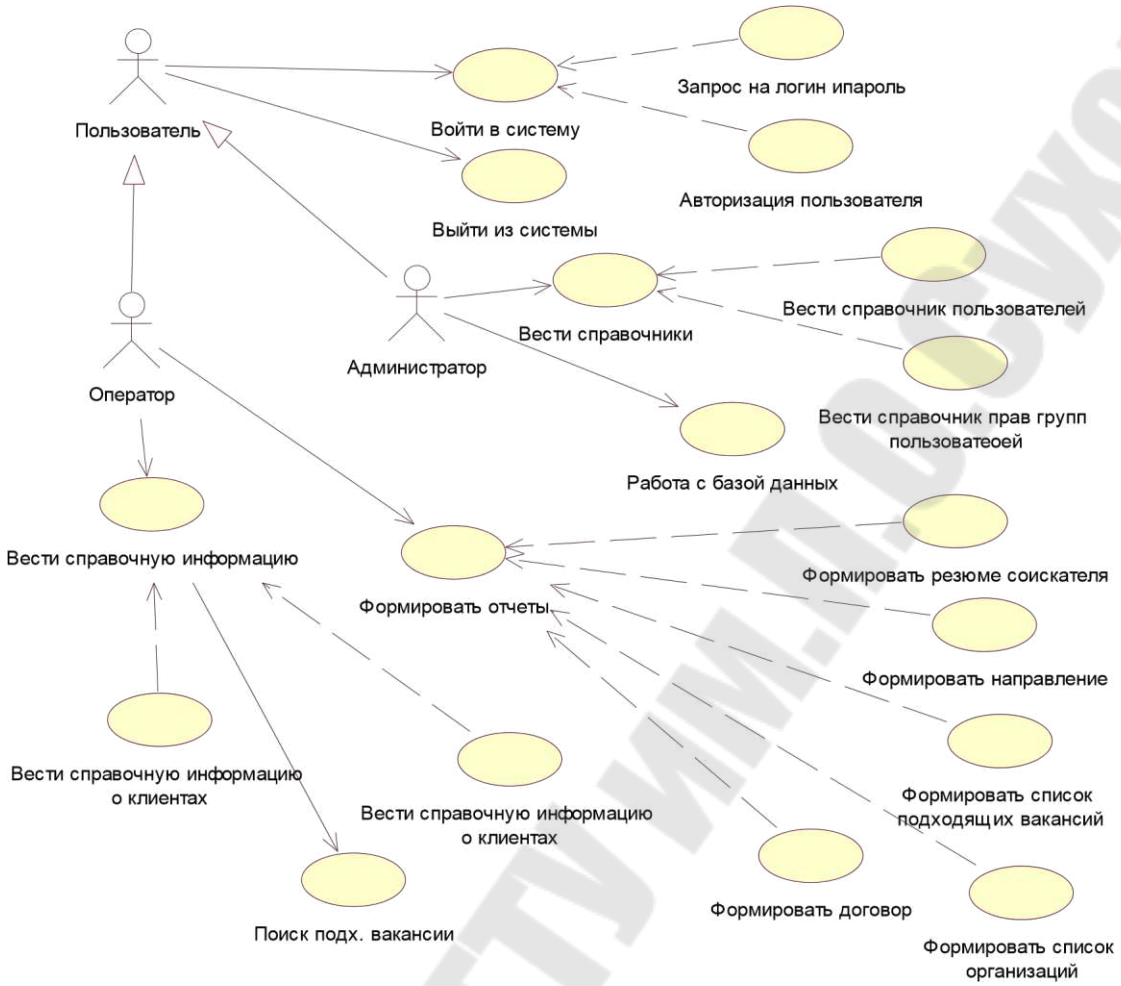

Рисунок 2.11 – Пример диаграммы вариантов использования

3. ТЕХНОЛОГИИ РАЗРАБОТКИ И ОТЛАДКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1. ИНСТРУМЕНТЫ ПРОГРАММИРОВАНИЯ И ОТЛАДКИ: ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ, СИСТЕМА КОНТРОЛЯ ВЕРСИЙ, СИСТЕМЫ УПРАВЛЕНИЯ ПРОЕКТАМИ

3.1.1 Интегрированная среда разработки ПО

Интегрированная среда разработки, ИСР (англ. *IDE*, *Integrated Development Environment* или *Integrated Debugging Environment*) –

система программных средств, используемая программистами для разработки программного обеспечения (ПО).

Обычно среда разработки включает в себя:

– Текстовый редактор;
– Компилятор и / или интерпретатор;
– Средства автоматизации сборки;
– Отладчик.

Иногда содержит также средства для интеграции с системами управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму иерархии классов – для использования при объектно-ориентированной разработке ПО. Хотя и существуют ИСР, предназначенные для нескольких языков программирования – такие, как *Eclipse*, *netbeans*, *Embarcadero RAD Studio*, *Qt Creator* или *Microsoft Visual Studio*, но обычно ИСР предназначается для одного определённого языка программирования – как, например, *Visual Basic*, *purebasic*, *Delphi*, *Dev-C++*.

Частный случай ИСР, их эволюционное развитие – среды визуальной разработки, которые включают в себя возможность визуального редактирования интерфейса программы.

Интегрированные среды разработки были созданы для того, чтобы максимизировать производительность программиста благодаря тесно связанным компонентам с простыми пользовательскими интерфейсами. Это позволит разработчику делать меньше действий для переключения различных режимов, в отличие от дискретных программ разработки. Однако, так как *IDE* является сложным программным комплексом, то лишь после долгого процесса обучения среда разработки сможет качественно ускорить процесс разработки ПО.

Обычно *IDE* ориентирована на определенный язык программирования, предоставляя набор функций, который наиболее близко соответствует парадигмам этого языка программирования. Однако, есть некоторые *IDE* с поддержкой нескольких языков, такие как *Eclipse*, *activestate Komodo*, последние версии *netbeans*, *Microsoft Visual Studio*, *windev* и *Xcode*.

IDE обычно представляет из себя единственную программу, в которой проводилась вся разработка. Она обычно содержит много функций для создания, изменения, компилирования, развертывания и

отладки программного обеспечения. Цель среды разработки заключается в том, чтобы абстрагировать конфигурацию, необходимую, чтобы объединить утилиты командной строки в одном модуле, который позволит уменьшить время, чтобы изучить язык, и повысить производительность разработчика. Также считается, что трудная интеграция задач разработки может далее повысить производительность. Например, *IDE* позволяет проанализировать код и тем самым обеспечить мгновенную обратную связь и уведомить о синтаксических ошибках. В то время как большинство современных *IDE* являются графическими, они использовались еще до того, как появились системы управления окнами (которые реализованы в *Microsoft Windows* или *X*11 для *unix*-систем). Они были основаны на тексте, используя функциональные клавиши или горячие клавиши, чтобы выполнить различные задачи (например, *Turbo Pascal*). Использование *IDE* для разработки программного обеспечения является прямой противоположностью способа, в котором используются несвязанные инструменты, такие как *vi* (текстовый редактор), *GCC* (компилятор), и т.п.

Интегрированные среды разработки также часто поддерживают пометки в комментариях в исходном тексте программ, отмечающие места, требующие дальнейшего внимания или предполагающие внесение изменений, такие как *TODO*. В дальнейшем эти пометки могут выделяться редакторами (напр. *Vim*, *emacs*, встроенный редактор *Visual Studio*) или использоваться для организации совместной работы с построением тегов и задач (например, в *intellij*). Использование комментариев с *TODO* так же является стандартом оформления кода на *Object Pascal*, *Delphi*. *Microsoft* в руководстве по *Visual Studio* рекомендует использовать тег *TODO* (наравне с *HACK*, *UNDONE*) для следующих пометок:

- Добавление новых функций;
- Известных проблем, которые нужно устранить;
- Предполагаемых к реализации классов;
- Мест размещения кода обработчиков ошибок;
- Напоминаний о необходимости переработки участка кода.

Обычно интегрированная среда разработки – это совокупность программных средств, поддерживающая все этапы разработки программного обеспечения от написания исходного текста программы до ее компиляции и отладки, и обеспечивающая простое и быстрое взаимодействие с другими инструментальными средствами

(программным отладчиком-симулятором, внутрисхемным эмулятором, эмулятором ПЗУ и программатором).

Строго говоря, интегрированные среды разработки не относятся к числу средств отладки. Отладка – лишь одно из свойств интегрированных сред, которые представляют собой основу любой визуальной среды разработки или *RAD*-среды.

При традиционном подходе, начальный этап написания программы строится следующим образом:

– Исходный текст набирается при помощи какого-либо текстового редактора.

– По завершении набора, работа с текстовым редактором прекращается и запускается кросс компилятор.

– Как правило, вновь написанная программа содержит синтаксические ошибки, и компилятор сообщает о них на консоль оператора.

– Вновь запускается текстовый редактор, и оператор должен найти и устранить выявленные ошибки, при этом сообщения о характере ошибок выведенные компилятором уже не видны, так как экран занят текстовым редактором.

И этот цикл может повторяться не один раз. Если программа имеет большой объем, собирается из различных частей, и подвергается длительному редактированию или модернизации, то даже этот начальный этап может потребовать много сил и времени. После этого наступает этап отладки программы и к редактору с компилятором добавляется эмулятор или симулятор, за работой которого хотелось бы следить прямо по тексту программы в текстовом редакторе.

Интегрированные среды (оболочки) разработки (*Integrated Development Environment*, *IDE*) позволяют избежать большого объема однообразных действий и тем самым существенно повысить эффективность процесса разработки и отладки позволяют, то есть они являются *RAD*-средами различной степени автоматизации процесса программирования.

Работа в интегрированной среде дает программисту:

– Возможность использования встроенного многофайлового текстового редактора, специально ориентированного на работу с исходными текстами программ;

– Иметь автоматическую диагностику выявленных при компиляции ошибок, когда исходный текст программы, доступный

редактированию, выводится одновременно с диагностикой в многооконном режиме;

– Возможность параллельной работы над несколькими проектами. Менеджер проектов позволяет использовать любой проект в качестве шаблона для вновь создаваемого проекта;

– Минимум перекомпиляции. Ей подвергаются только редактировавшиеся модули;

– Возможность загрузки отлаживаемой программы в имеющиеся средства отладки, и возможность работы с ними без выхода из оболочки;

– Возможность подключения к оболочке практически любых программных средств.

В последнее время, функции интегрированных сред разработки становятся стандартной принадлежностью программных интерфейсов эмуляторов и отладчиков-симуляторов.

Подобные функциональные возможности, в сочетании с дружественным интерфейсом, в состоянии существенно увеличить скорость разработки программ, особенно для микроконтроллеров и процессоров цифровой обработки сигналов, являющихся очень трудоемкими и труднообозримыми процессами.

3.1.2 Системы управления версиями

Система контроля версий – это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или *web*-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) – как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли

файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельный каталог (возможно даже, каталог с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть в каком каталоге вы находитесь и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

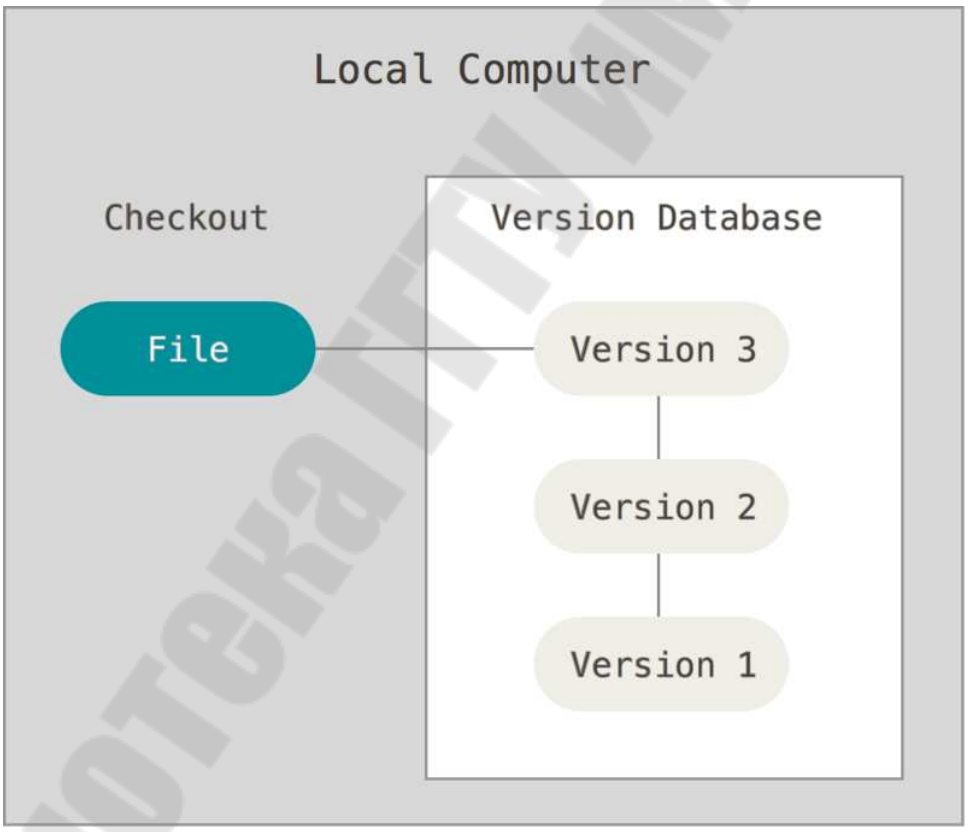

Рис. 3.1. Локальный контроль версий

Одной из популярных СКВ была система *RCS*, которая и сегодня распространяется со многими компьютерами. *RCS* хранит на диске наборы патчей (различий между файлами) в специальном

формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, – это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как *CVS*, *Subversion* и *Perforce*, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.

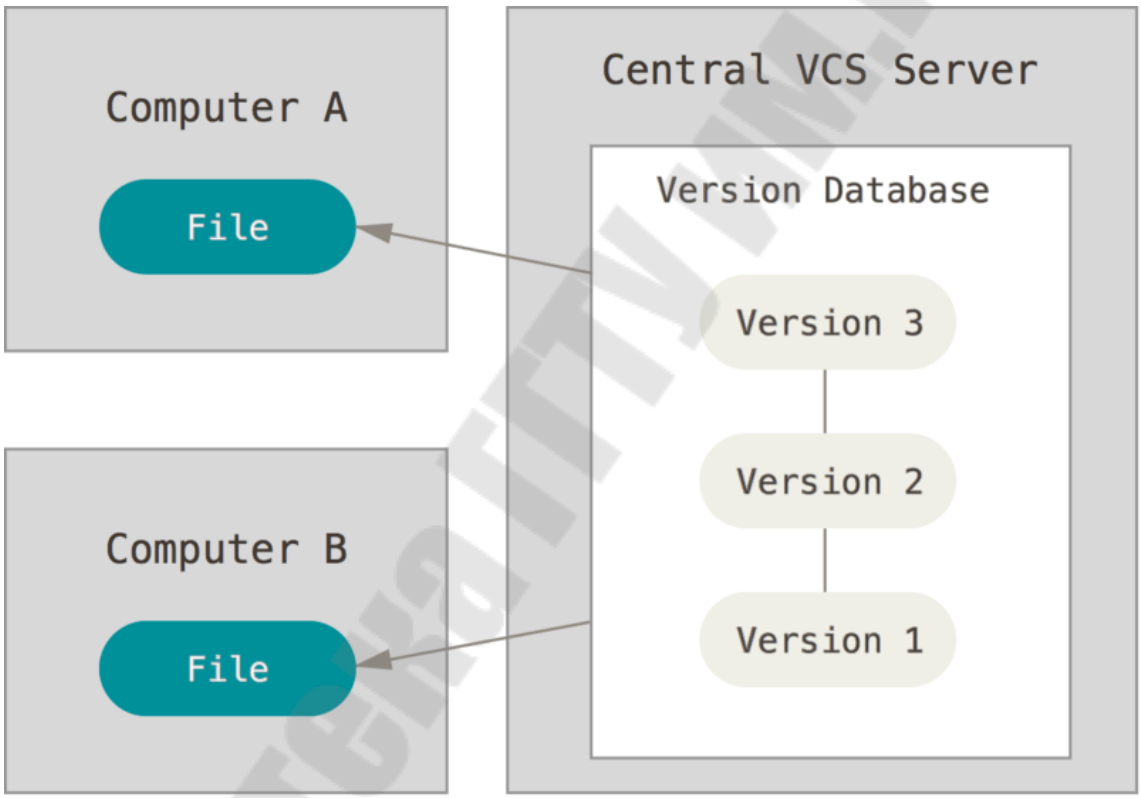

Рис. 3.2. Централизованный контроль версий

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус – это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё – всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как *Git*, *Mercurial*, *Bazaar* или *Darcs*) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

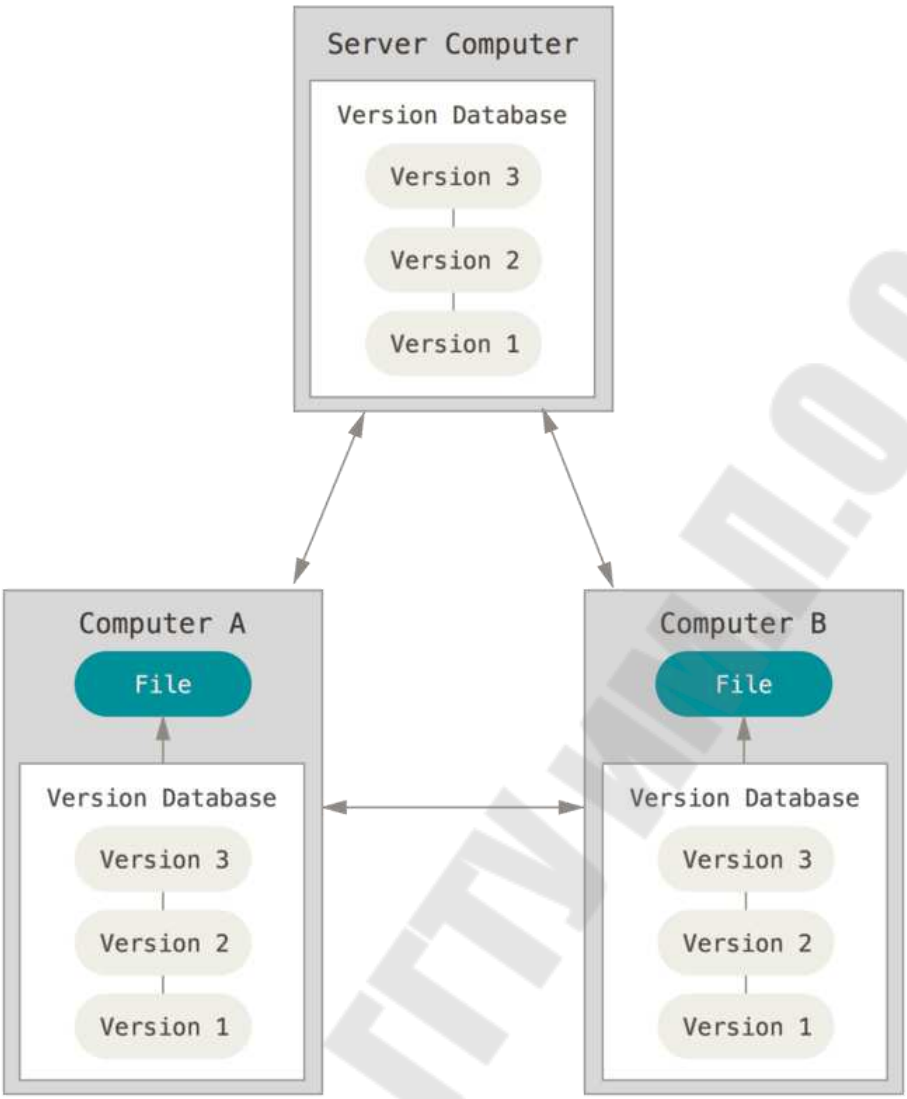

Рис. 3.3. Распределённый контроль версий

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

3.1.3 Работа с репозиториями

С момента своего появления в 2005 году, *Git* развился в простую в использовании систему, сохранив при этом свои изначальные

качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки.

Создание *Git*-репозитория

Обычно вы получаете репозиторий *Git* одним из двух способов:

Вы можете взять локальный каталог, который в настоящее время не находится под версионным контролем, и превратить его в репозиторий *Git*, либо Вы можете клонировать существующий репозиторий *Git* из любого места.

В обоих случаях вы получите готовый к работе *Git* репозиторий на вашем компьютере.

Создание репозитория в существующем каталоге

Если у вас уже есть проект в каталоге, который не находится под версионным контролем *Git*, то для начала нужно перейти в него. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному:

Для *Linux*:

$ *cd /home/user/my_project*

Для *macos*:

$ *cd /Users/user/my_project*

Для *Windows*:

$ *cd C:/Users/user/my_project*

А затем выполните команду:

$ *git init*

Эта команда создаёт в текущем каталоге новый подкаталог с именем *.git*, содержащий все необходимые файлы репозитория – структуру *Git* репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду *git add* несколько раз, указав индексируемые файлы, а затем выполнив *git commit*:

$ *git add *.c*

$ *git add LICENSE*

$ *git commit -m 'Initial project version'*

Мы разберем, что делают эти команды чуть позже. Теперь у вас есть *Git*-репозиторий с отслеживаемыми файлами и начальным коммитом.

Клонирование существующего репозитория

Для получения копии существующего *Git*-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду *git clone*. Если вы знакомы с другими системами контроля версий, такими как *Subversion*, то заметите, что команда называется «*clone*», а не «*checkout*». Это важное различие – вместо того, чтобы просто получить рабочую копию, *Git* получает копию практически всех данных, которые есть на сервере. При выполнении *git clone* с сервера забирается (*pulled*) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных хуков (*server-side hooks*) и т. П., но все данные, помещённые под версионный контроль, будут сохранены).

Клонирование репозитория осуществляется командой *git clone* <*url*>. Например, если вы хотите клонировать библиотеку *libgit*2, вы можете сделать это следующим образом:

$ *git clone https://github.com/libgit2/libgit2*

Эта команда создаёт каталог *libgit*2, инициализирует в нём подкаталог *.git*, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог *libgit*2, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в каталог с именем, отличающимся от *libgit*2, необходимо указать желаемое имя, как параметр командной строки:

$ *git clone https://github.com/libgit2/libgit2 mylibgit*

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван *mylibgit*.

В *Git* реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол *https://*, вы также можете встретить *git://* или *user@server:path/to/repo.git*, использующий протокол передачи *SSH*.

Запись изменений в репозиторий

Итак, у вас имеется настоящий *Git*-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (*snapshots*) этих

изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы – это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы – это те файлы, о которых знает *Git*.

Неотслеживаемые файлы – это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что *Git* только что их извлек, и вы ничего пока не редактировали.

Как только вы отредактируете файлы, *Git* будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

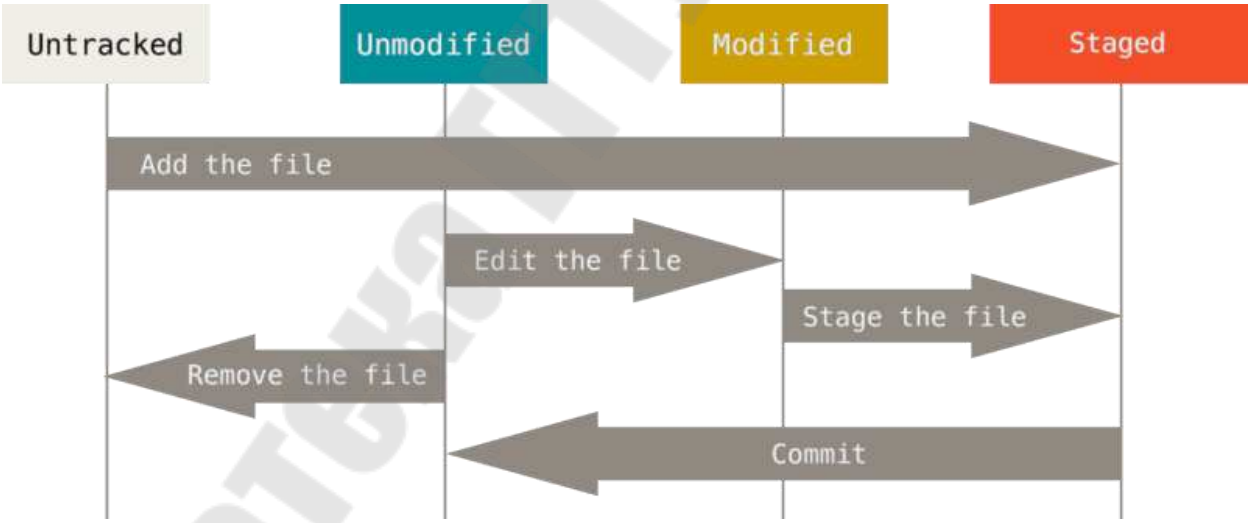

Рис. 3.4. Жизненный цикл состояний файлов

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся – это команда *git status*. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

$ *git status*

On branch master

Your branch is up-to-date with '*origin/master*'.

Nothing to commit, *working tree clean*

Это означает, что у вас чистый рабочий каталог, другими словами – в нем нет отслеживаемых измененных файлов. *Git* также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере. Пока что это всегда ветка *master*, ветка по умолчанию; в этой главе это не важно.

Предположим, вы добавили в свой проект новый файл, простой файл *README*. Если этого файла раньше не было, и вы выполните *git status*, вы увидите свой неотслеживаемый файл вот так:

$ *echo 'My Project' > README*

$ *git status*

On branch master

Your branch is up-to-date with '*origin/master*'.

Untracked files:

(*use* "*git add <file>...*" *to include in what will be committed*)

README

Nothing added to commit but untracked files present (*use* "*git add*" *to track*)

Понять, что новый файл *README* неотслеживаемый можно по тому, что он находится в секции «*Untracked files*» в выводе команды *status*. Статус *Untracked* означает, что *Git* видит файл, которого не было в предыдущем снимке состояния (коммите); *Git* не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить *README*, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда *git add*. Чтобы начать отслеживание файла *README*, вы можете выполнить следующее:

$ *git add README*

Если вы снова выполните команду *status*, то увидите, что файл *README* теперь отслеживаемый и добавлен в индекс:

$ *git status*

On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git restore --staged <file>..." to unstage*)
New file: *README*

Вы можете видеть, что файл проиндексирован, так как он находится в секции «*Changes to be committed*». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды *git add*, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили *git init*, затем вы выполнили *git add* (файлы) – это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда *git add* принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл *CONTRIBUTING.md* и после этого снова выполните команду *git status*, то результат будет примерно следующим:

$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
New file: *README*
Changes not staged for commit:
(*use "git add <file>..." to update what will be committed*)
(*use "git checkout -- <file>..." to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Файл *CONTRIBUTING.md* находится в секции «*Changes not staged for commit*» – это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду *git add*. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например, для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если

вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект». Выполним *git add*, чтобы проиндексировать *CONTRIBUTING.md*, а затем снова выполним *git status*:

$ *git add CONTRIBUTING.md*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
New file: *README*
Modified: *CONTRIBUTING.md*

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в *CONTRIBUTING.md* до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним *git status*:

$ *vim CONTRIBUTING.md*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
New file: *README*
Modified: *CONTRIBUTING.md*
Changes not staged for commit:
(*use "git add <file>..." to update what will be committed*)
(*use "git checkout -- <file>..." to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Теперь *CONTRIBUTING.md* отображается как проиндексированный и не проиндексированный одновременно. Такая ситуация наглядно демонстрирует, что *Git* индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду *git add*. Если вы выполните коммит сейчас, то файл *CONTRIBUTING.md* попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду *git add* , а не в том, в котором он находится в вашем рабочем каталоге

в момент выполнения *git commit*. Если вы изменили файл после выполнения *git add*, вам придётся снова выполнить *git add*, чтобы проиндексировать последнюю версию файла:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    New file:   README
    Modified:   CONTRIBUTING.md
```

Сокращенный вывод статуса

Вывод команды *git status* довольно всеобъемлющий и многословный. *Git* также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните *git status -s* или *git status --short* вы получите гораздо более упрощенный вывод:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Новые неотслеживаемые файлы помечены ?? Слева от них, файлы добавленные в отслеживаемые помечены *A*, отредактированные файлы помечены *M* и так далее. В выводе содержится два столбца – в левом указывается статус файла, а в правом модифицирован ли он после этого. К примеру, в нашем выводе, файл *README* модифицирован в рабочем каталоге, но не проиндексирован, а файл *lib/simplegit.rb* модифицирован и проиндексирован. Файл *Rakefile* модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т. П.). В таком случае, вы можете создать файл *.gitignore*. С

перечислением шаблонов соответствующих таким файлам. Вот пример файла *.gitignore*:

$ *cat .gitignore*

**.[oa]*

**~*

Первая строка предписывает *Git* игнорировать любые файлы заканчивающиеся на «*.o*» или «*.a*» – объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы, заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например, *Emacs*, для обозначения временных файлов. Вы можете также включить каталоги *log*, *tmp* или *pid*; автоматически создаваемую документацию; и т. Д. И т. П. Хорошая практика заключается в настройке файла *.gitignore* до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле *.gitignore* применяются следующие правила:

Пустые строки, а также строки, начинающиеся с #, игнорируются.

Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.

Чтобы избежать рекурсии используйте символ слеш (/) в начале шаблона.

Чтобы исключить каталог добавьте слеш (/) в конец шаблона.

Можно инвертировать шаблон, использовав восклицательный знак (!) В качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (*) соответствует 0 или более символам; последовательность *[abc]* – любому символу из указанных в скобках (в данном примере *a*, *b* или *c*); знак вопроса (?) Соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом (*[0-9]*), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звёздочки, чтобы указать на вложенные каталоги: *a/**/z* соответствует *a/z*, *a/b/z*, *a/b/c/z*, и так далее.

Вот ещё один пример файла *.gitignore*:

Исключить все файлы с расширение *.a*

*.a

Но отслеживать файл *lib.a* даже если он подпадает под исключение выше

!Lib.a

Исключить файл *TODO* в корневом каталоге, но не файл в *subdir/TODO*

/TODO

Игнорировать все файлы в каталоге *build/*

Build/

Игнорировать файл *doc/notes.txt*, но не файл *doc/server/arch.txt*

Doc/*.txt

Игнорировать все *.txt* файлы в каталоге *doc/*

Doc/**/*.txt

Просмотр индексированных и неиндексированных изменений

Если результат работы команды *git status* недостаточно информативен для вас – вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены – вы можете использовать команду *git diff*. Позже мы рассмотрим команду *git diff* подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь включить в коммит. Если *git status* отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, *git diff* показывает вам непосредственно добавленные и удалённые строки – патч как он есть.

Допустим, вы снова изменили и проиндексировали файл *README*, а затем изменили файл *CONTRIBUTING.md* без индексирования. Если вы выполните команду *git status*, вы опять увидите что-то вроде:

$ *git status*

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(*use "git reset HEAD <file>..." to unstage*)

Modified: *README*

Changes not staged for commit:

(*use "git add <file>..." to update what will be committed*)

(*use "git checkout -- <file>..." to discard changes in working directory*)

Modified: *CONTRIBUTING.md*

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите *git diff* без аргументов:

$ *git diff*

Diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

Index 8*ebb*991..643*e*24*f* 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ *branch directly, things can get messy.*

Please include a nice description of your changes when you submit your PR;

If we have to read the whole diff to figure out why you're contributing

In the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if you patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (*and note in the PR title that it's*

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить *git diff --staged*. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

$ *git diff --staged*

Diff --git a/README b/README

New file mode 100644

Index 0000000..03902*a*1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

Важно отметить, что *git diff* сама по себе не показывает все изменения, сделанные с последнего коммита – только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как

если вы проиндексируете все свои изменения, то *git diff* ничего не вернёт.

Другой пример: вы проиндексировали файл *CONTRIBUTING.md* и затем изменили его, вы можете использовать *git diff* для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Если наше окружение выглядит вот так:

$ *git add CONTRIBUTING.md*
$ *echo '# test line' >> CONTRIBUTING.md*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
Modified: *CONTRIBUTING.md*
Changes not staged for commit:
(*use "git add <file>..." to update what will be committed*)
(*use "git checkout -- <file>..." to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Используйте *git diff* для просмотра не проиндексированных изменений

$ *git diff*
Diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
Index 643*e*24*f*..87*f*08*c*8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ *at the*
Starter Projects
See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line

А так же *git diff --cached* для просмотра проиндексированных изменений (*--staged* и *--cached* синонимы):

$ *git diff --cached*
Diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
Index 8*ebb*991..643*e*24*f* 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ *branch directly, things can get messy.*

Please include a nice description of your changes when you submit your PR;

If we have to read the whole diff to figure out why you're contributing

In the first place, *you're less likely to get feedback and have your change*

-merged in.

+merged in. *Also*, *split your changes into comprehensive chunks if you patch is*

+longer than a dozen lines.

If you are starting to work on a particular area, *feel free to submit a PR*

That highlights your work in progress (*and note in the PR title that it's*

Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано – любые файлы, созданные или изменённые вами, и для которых вы не выполнили *git add* после редактирования – не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли *git status*, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения – это набрать *git commit*:

$ *git commit*

Эта команда откроет выбранный вами текстовый редактор.

В редакторе будет отображён следующий текст (это пример окна *Vim*):

 # Please enter the commit message for your changes. Lines starting
 # with '#' will be ignored, and an empty message aborts the commit.
 # On branch master
 # Your branch is up-to-date with 'origin/master'.
#
 # Changes to be committed:
 # new file: README
 # modified: CONTRIBUTING.md
#
~

~
~
".git/COMMIT_EDITMSG" 9L, 283C

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды *git status* и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Когда вы выходите из редактора, *Git* создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды *diff*.

Есть и другой способ – вы можете набрать свой комментарий к коммиту в командной строке вместе с командой *commit* указав его после параметра *-m*, как в следующем примере:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 Create mode 100644 README
```

Итак, вы создали свой первый коммит. Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (*master*), какая контрольная сумма *SHA*-1 у этого коммита (463*dc*4*f*), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, *Git* предоставляет простой способ. Добавление параметра *-a* в команду *git commit* заставляет *Git* автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без *git add*:

```
$ git status
```

On branch master

Your branch is up-to-date with '*origin/master*'.

Changes not staged for commit:

(*use* "*git add* <*file*>..." *to update what will be committed*)

(*use* "*git checkout* -- <*file*>..." *to discard changes in working directory*)

Modified: *CONTRIBUTING.md*

No changes added to commit (*use* "*git add*" *and/or* "*git commit -a*")

$ *git commit -a -m* '*Add new benchmarks*'

[master 83*e*38*c*7*] Add new benchmarks*

1 *file changed*, 5 *insertions*(+), 0 *deletions*(-)

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять *git add* для файла *CONTRIBUTING.md*, потому что флаг *-a* включает все файлы. Это удобно, но будьте осторожны: флаг *-a* может включить в коммит нежелательные изменения.

Удаление файлов

Для того чтобы удалить файл из *Git*, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда *git rm*, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как «неотслеживаемый».

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции «*Changes not staged for commit*» (измененные, но не проиндексированные) вывода команды *git status*:

$ *rm PROJECTS.md*

$ *git status*

On branch master

Your branch is up-to-date with '*origin/master*'.

Changes not staged for commit:

(*use* "*git add/rm* <*file*>..." *to update what will be committed*)

(*use* "*git checkout* -- <*file*>..." *to discard changes in working directory*)

Deleted: *PROJECTS.md*

No changes added to commit (*use* "*git add*" *and/or* "*git commit -a*")

Затем, если вы выполните команду *git rm*, удаление файла попадёт в индекс:

$ *git rm PROJECTS.md*

Rm '*PROJECTS.md*'

$ *git status*

On branch master
Your branch is up-to-date with '*origin/master*'.
Changes to be committed:
(*use* "*git reset HEAD <file>...*" *to unstage*)
Deleted: *PROJECTS.md*

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра *-f*. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из *Git*.

Другая полезная штука, которую вы можете захотеть сделать – это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл *.gitignore* и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --*cached*:

$ *git rm --cached README*

В команду *git rm* можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде:

$ *git rm log/*.log*

Обратите внимание на обратный слеш (\) перед *. Он необходим из-за того, что *Git* использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, имеющие расширение *.log* и находящиеся в каталоге *log*/. Или же вы можете сделать вот так:

$ *git rm* *~

Эта команда удаляет все файлы, имена которых заканчиваются на ~.

Перемещение файлов

В отличие от многих других систем версионного контроля, *Git* не отслеживает перемещение файлов явно. Когда вы переименовываете файл в *Git*, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован.

Таким образом, наличие в *Git* команды *mv* выглядит несколько странным. Если вам хочется переименовать файл в *Git*, вы можете сделать что-то вроде:

$ *git mv file_from file_to*

И это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что *Git* считает, что произошло переименование файла:

$ *git mv README.md README*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
Renamed: *README.md -> README*

Однако, это эквивалентно выполнению следующих команд:

$ *mv README.md README*
$ *git rm README.md*
$ *git add README*

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду *mv*. Единственное отличие состоит лишь в том, что *mv* – одна команда вместо трёх – это функция для удобства. Важнее другое – вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами *add*/*rm* перед коммитом.

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо *Git*-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду *git remote*. Она выведет названия доступных удалённых репозиториев. Если вы клонировали

репозиторий, то увидите, как минимум *origin* – имя по умолчанию, которое *Git* даёт серверу, с которого производилось клонирование:

$ *git clone https://github.com/schacon/ticgit*
Cloning into 'ticgit'...
Remote: *Reusing existing pack*: 1857, *done*.
Remote: *Total* 1857 (*delta* 0), *reused* 0 (*delta* 0)
Receiving objects: 100% (1857/1857), 374.35 *kib* | 268.00 *kib/s*, *done*.
Resolving deltas: 100% (772/772), *done*.
Checking connectivity... Done.
$ *cd ticgit*
$ *git remote*
Origin

Вы можете также указать ключ *-v*, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

$ *git remote -v*
Origin *https://github.com/schacon/ticgit* (*fetch*)
Origin *https://github.com/schacon/ticgit* (*push*)

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

$ *cd grit*
$ *git remote -v*
Bakkdoor *https://github.com/bakkdoor/grit* (*fetch*)
Bakkdoor *https://github.com/bakkdoor/grit* (*push*)
*Cho*45 *https://github.com/cho*45*/grit* (*fetch*)
*Cho*45 *https://github.com/cho*45*/grit* (*push*)
Defunkt *https://github.com/defunkt/grit* (*fetch*)
Defunkt *https://github.com/defunkt/grit* (*push*)
Koke *git://github.com/koke/grit.git* (*fetch*)
Koke *git://github.com/koke/grit.git* (*push*)
Origin *git@github.com:mojombo/grit.git* (*fetch*)
Origin *git@github.com:mojombo/grit.git* (*push*)

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи и в них можно отправлять свои

изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Обратите внимание на разнообразие протоколов, используемых при указании адреса удалённого репозитория.

Добавление удалённых репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (*shortname*), просто выполните команду *git remote add* <*shortname*> <*url*>:

$ *git remote*
Origin
$ *git remote add pb https://github.com/paulboone/ticgit*
$ *git remote -v*
Origin *https://github.com/schacon/ticgit* (*fetch*)
Origin *https://github.com/schacon/ticgit* (*push*)
Pb *https://github.com/paulboone/ticgit* (*fetch*)
Pb *https://github.com/paulboone/ticgit* (*push*)

Теперь вместо указания полного пути вы можете использовать *pb*. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду *git fetch pb*:

$ *git fetch pb*
Remote: *Counting objects*: 43, *done*.
Remote: *Compressing objects*: 100% (36/36), *done*.
Remote: *Total* 43 (*delta* 10), *reused* 31 (*delta* 5)
Unpacking objects: 100% (43/43), *done*.
From https://github.com/paulboone/ticgit
* *[new branch]* *master* -> *pb/master*
* *[new branch]* *ticgit* -> *pb/ticgit*

Ветка *master* из репозитория Пола сейчас доступна вам под именем *pb/master*. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола.

Получение изменений из удалённого репозитория – *Fetch* и *Pull*

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

$ *git fetch [remote-name]*

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда *clone* автоматически добавляет этот удалённый репозиторий под именем «*origin*». Таким образом, *git fetch origin* извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью *fetch*). Важно отметить, что команда *git fetch* забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду *git pull* чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда *git clone* автоматически настраивает вашу локальную ветку *master* на отслеживание удалённой ветки *master* на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение *git pull*, как правило, извлекает (*fetch*) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (*merge*) их с кодом, над которым вы в данный момент работаете.

Отправка изменений в удаленный репозиторий (*Push*)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: *git push* <*remote-name*> <*branch-name*>. Чтобы отправить вашу ветку *master* на сервер *origin* (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

$ *git push origin master*

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду *push*. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду *push*, а после

него выполнить команду *push* попытаетесь вы, то ваш *push* точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить *push*.

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду *git remote show* <*remote*>. Выполнив эту команду с некоторым именем, например, *origin*, вы получите следующий результат:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
   Master                       tracked
   Dev-branch                   tracked
  Local branch configured for 'git pull':
   Master merges with remote master
  Local ref configured for 'git push':
  Master pushes to master (up to date)
```

Она выдаёт *URL* удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке *master*, выполните *git pull*, ветка *master* с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации, и вы наверняка встречались с чем-то подобным. Однако, если вы используете *Git* более интенсивно, вы можете увидеть гораздо большее количество информации от *git remote show*:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
   Master                       tracked
```

Dev-branch *tracked*
Markdown-strip *tracked*
Issue-43 *new* (*next fetch will store in remotes*/*origin*)
Issue-45 *new* (*next fetch will store in remotes*/*origin*)
Refs/*remotes*/*origin*/*issue*-11 *stale* (*use* '*git remote prune*' *to remove*)
Local branches configured for '*git pull*':
Dev-branch merges with remote dev-branch
Master merges with remote master
Local refs configured for '*git push*':
Dev-branch *pushes to dev-branch* (*up to date*)
Markdown-strip *pushes to markdown-strip* (*up to date*)
Master *pushes to master* (*up to date*)

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении *git push*. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении *git pull*.

Удаление и переименование удалённых репозиториев

Для переименования, удалённого репозитория можно выполнить *git remote rename*. Например, если вы хотите переименовать *pb* в *paul*, вы можете это сделать при помощи *git remote rename*:

$ *git remote rename pb paul*
$ *git remote*
Origin
Paul

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как *pb*/*master*, теперь стало *paul*/*master*.

Если по какой-то причине вы хотите удалить удаленный репозиторий – вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения – вы можете использовать *git remote rm*:

$ *git remote remove paul*
$ *git remote*
Origin

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

***SSH* доступ**

Если у вас уже есть сервер, к которому все ваши разработчики имеют доступ по *SSH*, проще всего разместить ваш первый репозиторий там, поскольку вам не нужно практически ничего делать (как мы уже обсудили в предыдущем разделе). Если вы хотите более сложного управления правами доступа к вашим репозиториям, вы можете сделать это обычными правами файловой системы, предоставляемыми операционной системой вашего сервера.

Если вы хотите разместить ваши репозитории на сервере, где нет учётных записей для членов команды, которым требуются права на запись, то вы должны настроить доступ по *SSH* для них. Будем считать, что если у вас для этого есть сервер, то *SSH*-сервер на нем уже установлен и через него вы получаете доступ.

Есть несколько способов предоставить доступ всем участникам вашей команды. Первый – создать учётные записи для каждого, это просто, но может быть весьма обременительно. Вероятно, вы не захотите для каждого пользователя выполнять *adduser* (или *useradd*) и задавать временные пароли.

Второй способ – это создать на сервере пользователя '*git*', попросить всех участников, кому требуется доступ на запись, прислать вам открытый ключ *SSH* и добавить эти ключи в файл *~/.ssh/authorized_keys* в домашнем каталоге пользователя '*git*'. Теперь все будут иметь доступ к этой машине используя пользователя '*git*'. Это никак не повлияет на данные в коммите – пользователь, под которым вы соединяетесь с сервером по *SSH*, не воздействует на созданные вами коммиты.

Другой способ сделать это – настроить *SSH* сервер на использование аутентификации через *LDAP*-сервер или любой другой имеющийся у вас централизованный сервер аутентификации. Вы можете использовать любой механизм аутентификации на сервере и считать, что он будет работать для *Git*, если пользователь может получить доступ к консоли по *SSH*.

Генерация открытого *SSH* ключа

Как отмечалось ранее, многие *Git*-серверы используют аутентификацию по открытым *SSH*-ключам. Для того чтобы предоставить открытый ключ, каждый пользователь в системе должен его сгенерировать, если только этого уже не было сделано ранее. Этот процесс аналогичен во всех операционных системах. Сначала вам стоит убедиться, что у вас ещё нет ключа. По умолчанию пользовательские *SSH* ключи сохраняются в каталоге ~/*.ssh* домашнем каталоге пользователя. Вы можете легко проверить наличие ключа перейдя в этот каталог и посмотрев его содержимое:

$ *cd ~/.ssh*
$ *ls*
*Authorized_keys*2 *id_dsa known_hosts*
Config id_dsa.pub

Ищите файл с именем *id_dsa* или *id_rsa* и соответствующий ему файл с расширением *.pub*. Файл с расширением *.pub* – это ваш открытый ключ, а второй файл – ваш приватный ключ. Если указанные файлы у вас отсутствуют (или даже нет каталога *.ssh*), вы можете создать их используя программу *ssh-keygen*, которая входит в состав пакета *SSH* в системах *Linux*/*Mac*, а для *Windows* поставляется вместе с *Git*:

$ *ssh-keygen -o*
Generating public/private rsa key pair.
Enter file in which to save the key (*/home/schacon/.ssh/id_rsa*):
Created directory '*/home/schacon/.ssh*'.
Enter passphrase (*empty for no passphrase*):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
*D*0:82:24:8*e*:*d*7:*f*1:*bb*:9*b*:33:53:96:93:49:*da*:9*b*:*e*3
schacon@mylaptop.local

Сначала программа попросит указать расположение файла для сохранения ключа (*.ssh/id_rsa*), затем дважды ввести пароль для шифрования. Если вы не хотите вводить пароль каждый раз при использовании ключа, то можете оставить его пустым или использовать программу *ssh-agent*. Если вы решили использовать пароль для приватного ключа, то настоятельно рекомендуется использовать опцию *-o*, которая позволяет сохранить ключ в формате,

более устойчивом ко взлому методом подбора, чем стандартный формат.

Теперь каждый пользователь должен отправить свой открытый ключ вам или тому, кто администрирует *Git*-сервер (подразумевается, что ваш *SSH*-сервер уже настроен на работу с открытыми ключами). Для этого достаточно скопировать содержимое файла с расширением *.pub* и отправить его по электронной почте. Открытый ключ выглядит примерно так:

$ *cat ~/.ssh/id_rsa.pub*

Ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTG
K9Tjom/BWDSU
Gpl+nafzlhdtyw7hdi4yz5ew18jh4jw9jbhufrviqzm7xlelevf4h9lfx5qvkb
pppswg0cda3
Pbv7kOdJ/mtyblwxfcr+hao3fxritbqxix1nkhxphazsmcilq8v6rjsnaqwds
dmfvslvk/7XA
T3faojoasncm1q9x5+3V0Ww68/eifmb1zuufljqjkprrx88xypndvjynby6
vw/Pb0rwert/En
Mz+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nkatmikj
n2so1d01qratlmqvssbx
Nrrfi9wrf+M7Q== schacon@mylaptop.local

3.2. ПРИНЦИПЫ И ТЕХНОЛОГИИ СОЗДАНИЯ КАЧЕСТВЕННОГО КОДА В JAVA

3.2.1 Рефакторинг

Рефа́кторинг (англ. Refactoring), или перепроекти́рование кода, перераба́тка кода, равноси́льное преобразова́ние алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы[1][2]. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной

перестройке программы и улучшению её согласованности и чёткости.

Цели рефакторинга

Цель рефакторинга — сделать код программы более легким для понимания; без этого рефакторинг нельзя считать успешным.

Рефакторинг следует отличать от оптимизации производительности. Как и рефакторинг, оптимизация обычно тоже не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу.

С другой стороны, нужно отличать рефакторинг и от реинжиниринга, который осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют реинжиниринг.

Причины применения рефакторинга

Рефакторинг нужно применять постоянно при разработке кода. Основными стимулами его проведения являются следующие задачи:

1. Необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;
2. Необходимо исправить ошибку, причины возникновения которой сразу не ясны;
3. Преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

Признаки плохого кода

Во многом при рефакторинге лучше полагаться на интуицию, основанную на опыте. Тем не менее имеются некоторые видимые проблемы в коде (англ. Code smells), требующие рефакторинга:

1. Дублирование кода;
2. Длинный метод;
3. Большой класс;
4. Длинный список параметров;
5. «жадные» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. Избыточные временные переменные;
7. Классы данных;
8. Несгруппированные данные.

Рефакторинг кода

В программировании термин рефакторинг означает изменение исходного кода программы без изменения его внешнего поведения. В

экстремальном программировании и других гибких методологиях рефакторинг является неотъемлемой частью цикла разработки ПО: разработчики попеременно то создают новые тесты и функциональность, то выполняют рефакторинг кода для улучшения его логичности и прозрачности. Автоматическое юнит-тестирование позволяет убедиться, что рефакторинг не разрушил существующую функциональность.

Рефакторинг изначально не предназначен для исправления ошибок и добавления новой функциональности, он вообще не меняет поведение программного обеспечения и это помогает избежать ошибок и облегчить добавление функциональности. Он выполняется для улучшения понятности кода или изменения его структуры, для удаления «мёртвого кода» — всё это для того, чтобы в будущем код было легче поддерживать и развивать. В частности, добавление в программу нового поведения может оказаться сложным с существующей структурой — в этом случае разработчик может выполнить необходимый рефакторинг, а уже затем добавить новую функциональность.

Это может быть перемещение поля из одного класса в другой, вынесение фрагмента кода из метода и превращение его в самостоятельный метод или даже перемещение кода по иерархии классов. Каждый отдельный шаг может показаться элементарным, но совокупный эффект таких малых изменений в состоянии радикально улучшить проект или даже предотвратить распад плохо спроектированной программы.

Методы рефакторинга

Наиболее употребимые методы рефакторинга:

- Изменение сигнатуры метода (change method signature)
- Инкапсуляция поля (encapsulate field)
- Выделение класса (extract class)
- Выделение интерфейса (extract interface)
- Выделение локальной переменной (extract local variable)
- Выделение метода (extract method)
- Генерализация типа (generalize type)
- Встраивание (inline)
- Введение фабрики (introduce factory)
- Введение параметра (introduce parameter)
- Подъём метода (pull up method)
- Спуск метода (push down method)

- Переименование метода (rename method)
- Перемещение метода (move method)
- Замена условного оператора полиморфизмом (replace conditional with polymorphism)
- Замена наследования делегированием (replace inheritance with delegation)
- Замена кода типа подклассами (replace type code with subclasses)

Изменение сигнатуры метода (change method signature)

Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.

Инкапсуляция поля (encapsulate field)

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется «Перемещение метода».

Выделение метода (extract method)

Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

Перемещение метода (move method)

Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.

Замена условного оператора полиморфизмом (replace conditional with polymorphism)

Условный оператор с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.

Основные принципы:

- Вначале следует создать базовый класс и нужное число подклассов;
- В некоторых случаях следует провести оптимизацию условного оператора путём «Выделения метода»;
- Возможно использование «Перемещения метода», чтобы поместить условный оператор в вершину иерархии наследования;
- Выбрав один из подклассов, нужно конкретизировать в нём полиморфный метод базового класса и переместить в него тело соответствующей ветви условного оператора;
- Повторить предыдущее действие для каждой ветви условного оператора;
- Заменить весь условный оператор вызовом полиморфного метода базового класса.

Проблемы, возникающие при проведении рефакторинга

- Проблемы, связанные с базами данных;
- Проблемы изменения интерфейсов;
- Трудности при изменении дизайна.

3.2.2 Исключения. Обработка исключительных ситуаций

В этой лекции обсуждается используемый в Java механизм *обработки исключений.* Исключение в Java — это объект, который описывает исключительное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключительное состояние, создается объект класса Exception. Этот объект пересылается в метод, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях.

Основы

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: — try, catch, throw, throws и finally. Схема работы этого механизма следующая. Вы пытаетесь (try) выполнить блок кода, и если при этом возникает ошибка, система возбуждает (throw) исключение, которое в зависимости от его типа вы можете перехватить (catch) или передать умалчиваемому (finally) обработчику.

Ниже приведена общая форма блока обработки исключений.

```
Try {
// блок кода }
Catch (типисключения1 e) {
// обработчик исключений типа типисключения1 }
catch (типисключения2 e) {
// обработчик исключений типа типисключения2
 Throw(e)   // повторное возбуждение исключения }
finally {
}
```

Типы исключений

В вершине иерархии исключений стоит класс Throwable. Каждый из типов исключений является подклассом класса Throwable. Два непосредственных наследника класса Throwable делят иерархию подклассов исключений на две различные ветви. Один из них — класс Exception — используется для описания исключительных ситуации, которые должны перехватываться программным кодом пользователя. Другая ветвь дерева подклассов Throwable — класс Error, который предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

Неперехваченные исключения

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. Например, очередная наша программа содержит выражение, при вычислении которого возникает деление на нуль.

```
Class Exc0 { public static void
main(string args[]) { int d = 0;
int a = 42 / d;
} }
```

Вот вывод, полученный при запуске нашего примера.

C:\> java Exc0

Java.lang.arithmeticexception: / by
zero at Exc0.main(Exc0.java:4)

Обратите внимание на тот факт что типом возбужденного исключения был не Exception и не Throwable. Это подкласс класса Exception, а именно: arithmeticexception, поясняющий, какая ошибка возникла при выполнении программы. Вот другая версия того же класса, в которой возникает та же исключительная ситуация, но на этот раз не в программном коде метода main.

```
Class Exc1 {
static void
subroutine() {  int
d = 0; int a = 10 /
d;
}
Public static void main(String args[]) {
Exc1.subroutine();
} }
```

Вывод этой программы показывает, как обработчик исключений исполняющей системы Java выводит содержимое всего стека вызовов.

C:\> java Exc1

Java.lang.arithmeticexception: / by
zero at
Exc1.subroutine(Exc1.java:4)
At Exc1.main(Exc1.java:7)

Try и catch

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово try. Сразу же после try-блока помещается блок catch, задающий тип исключения которое вы хотите обрабатывать.

```
Class Exc2 { public static void
main(String args[]) {  try {     int
d = 0;
    Int a = 42 / d;
   }
Catch (arithmeticexception e) {
System.out.println("division by zero");
}
} }
```

Целью большинства хорошо сконструированных catch-разделов должна быть обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние — такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было (в нашем примере выводится предупреждение – division by zero).

Несколько разделов catch

В некоторых случаях один и тот же блок программного кода может возбуждать исключения различных типов. Для того, чтобы обрабатывать подобные ситуации, Java позволяет использовать любое количество catchразделов для try-блока. Наиболее специализированные классы исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел catch общего назначения, перехватывающий все подклассы класса Throwable.

```
Class multicatch {
Public static void main(String
args[]) {  try {
    Int a = args.length;
    System.out.println("a = " + a);
    Int b = 42
/ a;      int c[]
= { 1 };
    C[42] = 99;
   }
```

```
Catch (arithmeticexception e) {
System.out.println("div by 0: " + e);
}
Catch(arrayindexoutofboundsexception e) {
System.out.println("array index oob: " + e);
}
} }
```

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на нуль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив a в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива arrayindexoutof Bounds. Ниже приведены результаты работы этой программы, запущенной и тем и другим способом.

C:\> java multicatch *a = 0 div by 0:*
java.lang.arithmeticexception: / by zero
C:\> java multicatch 1 *a = 1 array index oob:*
java.lang.arrayindexoutofboundsexception: 42

Операторы try можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора try низкого уровня нет раздела catch, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы catch внешнего оператора try. Вот пример, в котором два оператора try вложены друг в друга посредством вызова метода.

Вложенные операторы try

```
class multinest {
static void
procedure() {
Try {     int
c[] = { 1 };
    C[42] = 99;
}
Catch(arrayindexoutofboundsexception e) {
```

```
System.out.println("array index oob: " + e);
} }
Public static void main(String
args[]) {  try {
    Int a = args.length();
    System.out.println("a = " + a);
    Int b = 42 / a;
    Procedure();
}
Catch (arithmeticexception e) {
System.out.println("div by 0: " + e);
}
} }
```

Throw

Оператор throw используется для возбуждения исключения «вручную».

Для того, чтобы сделать это, нужно иметь объект подкласса класса Throwable, который можно либо получить как параметр оператора catch, либо создать с помощью оператора new. Ниже приведена общая форма оператора throw.

Throw объектmmunathrowable;

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор throw возбуждает исключительную ситуацию, после чего то же исключение возбуждается повторно — на этот раз уже кодом перехватившего его в первый раз раздела catch.

```
Class throwdemo {
static void
demoproc() {
Try {
Throw new nullpointerexception("demo");
}
Catch (nullpointerexception e) {
System.out.println("caught inside demoproc");
throw e;
} }
Public static void main(String
args[]) {  try {
Demoproc();
}
Catch(nulpointerexception e) {
System.out.println("recaught: " + e);
}
} }
```

В этом примере обработка исключения проводится в два приема. Метод main создает контекст для исключения и вызывает demoproc. Метод demoproc также устанавливает контекст для обработки исключения, создает новый объект класса nullpointerexception и с помощью оператора throw возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода demoproc, причем объект-исключение доступен коду обработчика через параметр e. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора throw, в результате чего оно передается обработчику исключений в методе main. Ниже приведен результат, полученный при запуске этого примера.

```
C:\> java throwdemo
Caught inside demoproc recaught:
java.lang.nullpointerexception: demo
```

Throws

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений. Для

задания списка исключений, которые могут возбуждаться методом, используется ключевое слово throws. Если метод в явном виде (т.е. С помощью оператора throw) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе throws в объявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен быть расширен следующим образом:

Тип имя_метода(список аргументов) throws список_исключений {}

Ниже приведен пример программы, в которой метод procedure пытается возбудить исключение, не обеспечивая ни программного кода для его перехвата, ни объявления этого исключения в заголовке метода. Такой программный код не будет оттранслирован.

```
Class throwsdemo1 {
static void
procedure() {
System.out.println("inside procedure");
Throw new illegalaccessexception("demo");
}
Public static void main(String args[]) {
Procedure();
} }
```

Для того, чтобы мы смогли оттранслировать этот пример, нам придется сообщить транслятору, что procedure может возбуждать исключения типа illegalaccessexception и в методе main добавить код для обработки этого типа исключений :

```
Class throwsdemo { static void procedure()
throws illegalaccessexception {
System.out.println(" inside procedure");
Throw new illegalaccessexception("demo");
}
Public static void main(String
args[]) {  try {
Procedure();
}
Catch (illegalaccessexception e) {
```

```
System.out.println("caught " + e);
}
} }
```

Ниже приведен результат выполнения этой программы.

C:\> java throwsdemo *inside procedure*
caught java.lang.illegalaccessexception:
demo

Finally

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово finally. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела catch, блок finally будет выполнен до того, как управление перейдет к операторам, следующим за разделом try. У каждого раздела try должен быть по крайней мере или один раздел catch или блок finally. Блок finally очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела finally.

```
Class finallydemo {
static void proca() {
Try {
System.out.println("inside proca");
Throw new runtimeexception("demo");
}
fi
n
al
ly
{
System.out.println("proca's finally");
} }
Static void procb() {
Try {
```

```
System.out.println("inside
procb"); return; } finally {
System.out.println("procb's finally");
} }
Public static void main(String
args[]) { try {
Proca();
}
Catch (Exception e) {}
Procb();
} }
```

В этом примере в методе proca из-за возбуждения исключения происходит преждевременный выход из блока try, но по пути «наружу» выполняется раздел finally. Другой метод procb завершает работу выполнением стоящего в try-блоке оператора return, но и при этом перед выходом из метода выполняется программный код блока finally. Ниже приведен результат, полученный при выполнении этой программы.

```
C:\> java
finallydemo
inside proca
proca's finally
inside procb
Procb's finally
```

Подклассы Exception

Только подклассы класса Throwable могут быть возбуждены или перехвачены. Простые типы — int, char и т.п., а также классы, не являющиеся подклассами Throwable, например, String и Object, использоваться в качестве исключений не могут. Наиболее общий путь для использования исключений — создание своих собственных подклассов класса Exception. Ниже приведена программа, в которой объявлен новый подкласс класса Exception.

```
Class myexception extends Exception {
```

```
Private int detail;
myexception(int a) {
detail = a:
}
Public String tostring() {
Return "myexception[" + detail + "]";
}

}

Class exceptiondemo { static void
compute(int a) throws myexception {
System.out.println("called computer + a
+ ")."); if (a > 10)
Throw new myexception(a);
System.out.println("normal exit.");
}
Public static void main(String
args[]) { try {
Compute(1);
Compute(20);
}
Catch (myexception e) {
System.out.println("caught" + e);
}
}

}
```

Этот пример довольно сложен. В нем сделано объявление подкласса myexception класса Exception. У этого подкласса есть специальный конструктор, который записывает в переменную объекта целочисленное значение, и совмещенный метод tostring, выводящий значение, хранящееся в объекте-исключении. Класс exceptiondemo определяет метод compute, который возбуждает исключение типа myexcepton. Простая логика метода compute возбуждает исключение в том случае, когда значение пара-ветра метода больше 10. Метод main в защищенном блоке вызывает метод compute сначала с допустимым значением, а затем — с недопустимым (больше 10), что позволяет

продемонстрировать работу при обоих путях выполнения кода. Ниже приведен результат выполнения программы.

```
C:\> java exceptiondemo
called compute(1).
Normal exit.
Called
compute(20).
Caught myexception[20]
```

Заключительное резюме

Обработка исключений предоставляет *исключительно* мощный механизм для управления сложными программами. Try, throw, catch дают вам простой и ясный путь для встраивания обработки ошибок и прочих нештатных ситуаций в программную логику. Если вы научитесь должным образом использовать рассмотренные в данной лекции механизмы, это придаст вашим классам профессиональный вид, и любые будущие пользователи вашего программного кода, несомненно, оценят это.

3.3. ПРИНЦИПЫ И ТЕХНОЛОГИИ СОЗДАНИЯ ДРУЖЕСТВЕННЫХ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ В JAVA

Построение графического интерфейса GUI в Java кажется достаточно простым. Но простота оказывается не такой простой, если учесть, что имеются две отдельных, но тесно связанных между собой библиотеки графических классов: Abstract Windows Toolkit (AWT) и Java Foundation Classes (JFC), которая известна как Swing. AWT построен на основе графической системы каждой платформы, на которой выполняется JVM. Например, когда создается кнопка на Windows, AWT создает контрольный элемент из графической библиотеки Windows. Чтобы избавиться от такой зависимости, была разработана библиотека на самой Java. Именно она называется Java Foundation Classes (Swing). AWT осталась для совместимости. Разработчики Java быстро поняли преимущества классов Swing и стали широко их использовать. Разработчики апплетов, однако, еще долго использовали AWT из-за того, что браузеры не поддерживали Swing.

В результате при написании апплетов приходится рассматривать три ситуации: браузеры с запрещенной Java, браузеры с установленной Java

1.1 и браузеры с Java 2. Часто разработчики просто пишут HTML-страницы без использования Java. Другая группа программистов требует, чтобы была установлена Java 2 с поддержкой Java. Обычно такой подход можно использовать только для корпоративных приложений, когда администраторы могут контролировать пользовательские компьютеры. Но так как примерно 90% Web-приложений используются неконтролируемыми пользователями, приходится до сих пор применять AWT.

Контейнер – это объект, который может содержать другие графические объекты. В этой лекции рассматриваются вопросы использования AWT контейнеров. Контейнеры могут помещаться в другие контейнеры и т. Д. Основные классы контейнеров:

- Applet— это контейнер для использования в браузере.
- Frame— это окно верхнего уровня с заголовком и границей.
- Panel— это прямоугольник, в который можно помещать другие компоненты, в том числе и другие объекты Panel.
- Scrollpane— это панель, в которой автоматически реализован вертикальный и горизонтальный скроллинг.
- Dialog— это диалоговое окно.

Рассмотрим эти контейнеры, а затем компоненты, которые могут размещаться в них.

Апплеты (Applet)

Сейчас уже трудно поверить, что при своем создании Java, в основном, была предназначена именно для создания апплетов. Java-простой объектноориентированный язык, который легко было использовать для создания многоплатформенных приложений. Сейчас, конечно, больше используется Java для создания приложений на стороне сервера, и многие программисты никогда не писали апплеты. Но есть преимущества использования апплетов. Они, несомненно, более мощные, чем javascript и HTML. Кроме того, они передаются через интернет, и их не надо хранить на клиентской машине.

С другой стороны, менеджер безопасности, поставляемый с JVM, ограничивает доступ к информации на пользовательской машине. Это приводит к уменьшению возможностей апплета. Также необходимо создавать апплеты так, чтобы их размер был как можно меньше, потому что они передаются по сети.

На самом деле при создании апплета используется базовый класс java.applet.Applet, который обеспечивает всю функциональность, необходимую для связи с браузером и его виртуальной машиной (JVM), поэтому написание апплета выполняется очень просто: необходимо

переопределить некоторые методы, чтобы апплет соответствовал предъфвлфемым требованиям. Класс java.applet.Applet есть в JVM, которая является частью браузера. Когда браузер получает файл HTML с Web-сервера, он находит тэг applet и выгружает требуемый файл .class для указанного апплета. Апплет должен наследовать класс java.applet.Applet содержать переопределенные методы. JVM будет обращаться к этим методам и, соответственно, вызывать переопределенные.

Запуск апплетов

Напишем простой апплет в несколько строк :

```
Import java.awt.*;
Import java.applet.Applet;

Public class helloapplet extends Applet
{
  Public void paint(Graphics g)
  {
    G.drawstring("Hello, Applet", 10, 10);
  } } import
java.applet.Applet;
```

Предупреждение:

Для того, чтобы использовать апплет, в HTML-файл вставляется тег – ссылка на апплет:

```
This is the Hello Applet
<applet
code=helloapplet
width=200
height=200>
</applet>
```

Тег <applet> указывает, что апплет с именем helloapplet.class должен быть выгружен с того же сервера, откуда и сам HTML-файл. Размер области, занятой апплетом, указывается в параметрах width и height. Для запуска htmlфайла с апплетом можно воспользоваться разными способами. Во-первых, можно просто запустить браузер и указать в строке адреса полный путь к файлу:

C:\runhelloapplet.html

Если на компьютере установлен Web-сервер, то нужно помесить и htmlфайл и класс апплета в директорий Web-сервера и указать в строке адреса браузера:

Http://127.0.0.1/runhelloapplet.html

В любом случае запустится файл runhelloapplet.html в браузере (рис. 7.1):

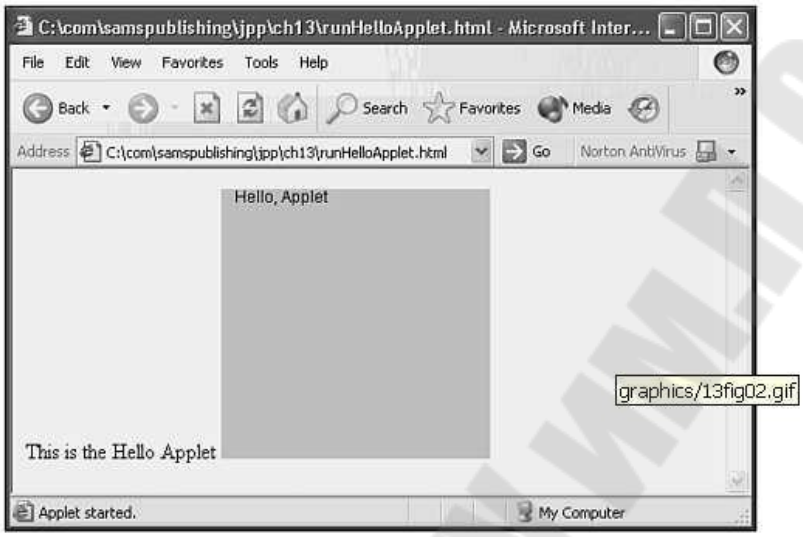

Рис. 7.1 – Вид апплета в браузере

Серая часть страницы – область апплета. Фраза "This is the Hello Applet" помещена кодом HTML. А фраза "Hello, Applet" выводится в окно браузера кодом апплета. Хотя апплеты и не самое важное в Java сейчас, с их помощью можно создавать интересные Web-страницы.

Фреймы (Frame)

Класс Frame – контейнер, который чаще всего используется для создания приложений на Java. Даже при создании Panels и scrollpanes их обычно помещают в объект Frame. Создать объект типа Frame можно двумя способами. Первый – создание производного от Frame класса. Второй – объявить объект Frame в методе main():

```
Import java.awt.*;
Import java.awt.event.*;

Public class frameextender extends Frame
{

  /** Конструктор */
```

```
  Public frameextender()
  {
    Addwindowlistener(new wincloser());
    Settitle("Just a Frame");
setbounds( 100, 100, 200, 200);
    Setvisible(true);
  }

  Public static void main(String args[])
  {
    Frameextender fe = new frameextender();
  }
}

Class wincloser extends windowadapter
{
  Public void windowclosing(windowevent e)
  {
    System.exit(0);
  }
}
```

В этом примере расширяется класс Frame. При этом вся функциональность класса наследуется. Public class frameextender extends Frame

В отличие от апплета, приложения не имеют метода init(). В конструкторе выполняются все действия по инициализации приложения. Public frameextender()

Для того, чтобы окна закрывались корректно, с очисткой всех необходимых ресурсов, необходимо добавить класс windowlistener. Swing обеспечивает более элегантный способ, но об этом будет речь позже.

Addwindowlistener(new wincloser());

Заголовок окна добавляется методом:

settitle("Just a Frame");

Размер начального окна устанавливается следующим методом.

Setbounds(100, 100, 200, 200);

Затем окно делается видимым:

```
Setvisible(true);
```

В метод main() создается объект типа frameextender, при этом выполняется конструктор.

```
  Public static void main(String args[])
  {
    Frameextender fe = new frameextender();
  }
}
```

Класс wincloser наследует класс windowapapter. Класс windowadapter – это абстрактный класс, который обеспечивает набор методов, выполняющих действия по умолчанию. Программист задает собственную реализацию переопределенных методов. Мы должны обеспечить корректное закрытие окна. Более подробно об обработке событий рассказывается в лекции 14. А сейчас просто выполняется закрытие окна.

```
Class wincloser extends windowadapter
{
  Public void windowclosing(windowevent e)
```

Метод exit() класса System приводит к завершению задачи и освобождению ресурсов приложения.

```
System.exit(0);
```

Чтобы запустить пример, откомпилируйте его, а затем наберите командную строку:

```
C:\>java frameextender
```

Вы увидите на экране (рис. 7.2):

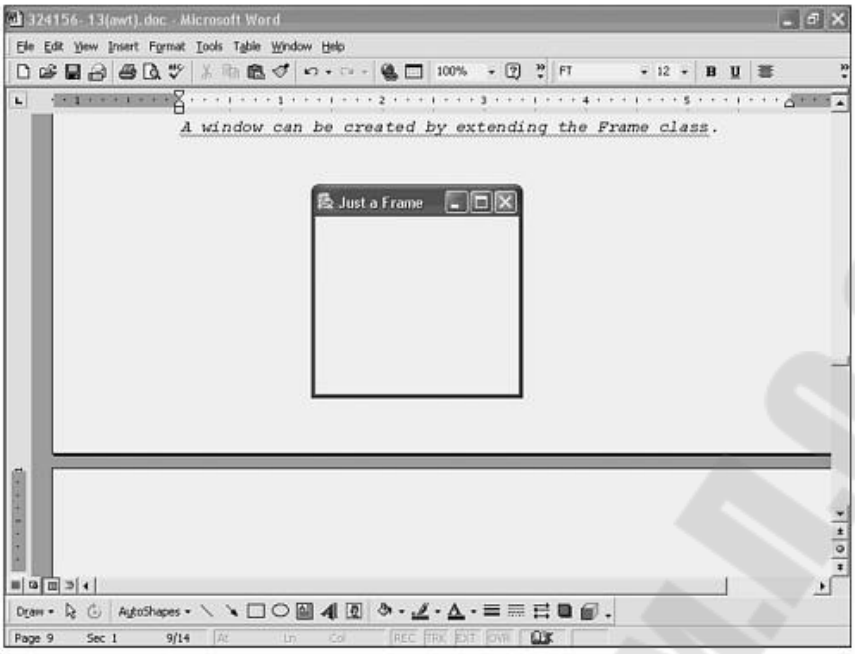

Рис. 7.2 – Вид графического приложерия с использованием класса Frame

Другой способ создания окна показан в листинге. Здесь объект типа Frame создается в методе main() создаваемого класса frameinstantiater:

```
Import java.awt.*;
Import java.awt.event.*;

Public class frameinstantiater
{
  Public frameinstantiater()
  {
  }

  Public static void main(String[] args)
  {
    Frame frame1 = new Frame();
frame1.addwindowlistener(new wincloser());
frame1.settitle("An Instantiated Frame");
frame1.setbounds( 100, 100, 300, 300);
    Frame1.setvisible(true);
```

```
  }
}

Class wincloser extends windowadapter
{
  Public void windowclosing(windowevent e)
  {
    System.exit(0);
  }
}
```

Здесь не выполняется расширение других

классов. Public class frameinstantiater

Конструктор пустой.

```
Public frameinstantiater()
{
}
```

Все основные действия выполняются в методе main. Вначале создается объект типа Frame:

```
Frame frame1 = new Frame();
```

Затем добавляется обработчик закрытия окна:

```
frame1.addwindowlistener(new wincloser());
```

Потом добавляется заголовок, устанавливаются размеры, и окно делается видимым.

```
    Frame1.settitle("An Instantiated Frame");
frame1.setbounds( 100, 100, 300, 300);
    Frame1.setvisible(true);
  }
}
```

Для запуска программы нужно ее скомпилировать и запустить, как было показано выше, естественно заменив имя файла и класса. Но фреймы бесполезны, если в них не размещаются элементы интерфейса. Самый простой способ разместить элементы в фрейме – это использовать метод add() класса Frame:

```
Import java.awt.*;
Import java.awt.event.*;
```

```
Public class textfieldinstantiater
{
  Public textfieldinstantiater()
  {
  }
  Public static void main(String[] args)
  {
    Frame frame1 = new Frame();
    Textfield tf1 = new textfield("Directly on the
Frame");       textfield tf2 = new textfield("On top of
the old text");       frame1.add(tf1);
frame1.add(tf2);

    Frame1.addwindowlistener(new wincloser());
frame1.settitle("An Instantiated Frame");
frame1.setbounds( 100, 100, 300, 300);
    Frame1.setvisible(true);
  }
}

Class wincloser extends windowadapter
{
  Public void windowclosing(windowevent e)
  {
    System.exit(0);

}

}
```

Новое в этом листинге – вставка двух текстовых полей:

```
Textfield tf1 = new textfield("Directly on the Frame");
textfield tf2 = new textfield("On top of the old text");
frame1.add(tf1);
Frame1.add(tf2);
```

Панели (Panel)

Класс java.awt.Panel – это прямоугольный контейнер, но без заголовка и границ. По умолчанию в него помещаются элементы слева направо, с переносом на новую строку. Такое заполнение обеспечивает менеджер компоновки flowlayout, который мы будем рассматривать позже. Панели используются, чтобы иметь несколько контейнеров внутри фрейма. Это дает большую гибкость для расположения элементов. В листинге, как дополнительная панель изменяет компоновку объектов:

```
/*
 * testpanel.java

Import java.awt.*;
Import java.awt.event.*;

Public class testpanel extends Frame
{
  Textfield tf1;
  Textfield tf2;

  /**Создание новой testpanel */
  Public testpanel()
  {
    Tf1 = new textfield("Directly on the Panel");
tf2 = new textfield("Following the first textfield");
    Panel p1 = new Panel();
    P1.add(tf1);
p1.add(tf2);
    Add(p1);
addwindowlistener(new wincloser());
    Settitle("Using a Panel");
setbounds( 100, 100, 300, 300);
    Setvisible(true);
  }

  Public static void main(String[] args)
  {
    Testpanel tp = new testpanel();
  }

}
```

```
Class wincloser extends windowadapter
{
  Public void windowclosing(windowevent e)
  {
    System.exit(0);
  }
}
```

Основное различие между этим примером и примером в листинге 13.5 заключается в том, что оба текстовых поля добавлены в панель, а панель вставляется в фрейм.

Оба текстовых поля инициализируются строками:

```
Tf1 = new textfield("Directly on the Panel");
tf2 = new textfield("Following the first
textfield");
```

Сождается новая панель, в нее добавляются текстовые поля:

```
Panel p1 = new Panel();
P1.add(tf1);
p1.add(tf2);
```

Панель вставляется в фрейм.

```
Add(p1);
```

Другой полезный контейнер - java.awt.scrollpane. Этот контрейнер позволяет вставить в него только один объект. Но этим объектом может быть другая панель, в которой находится множество элементов. Отличие этого контейнера от других – наличие горизонтальной и вертикальной полос прокрутки. Задать присутствие этих полос можно с помощью конструктора, передавая один из параметров:

- Scrollpane.SCROLLBARS_AS_NEEDED (когда необходимо)
- Scrollpane.SCROLLBARS_ALWAYS (всегда)
- Scrollpane.SCROLLBARS_NEVER (никогда) По умолчанию действует значение Scroll-

Pane.SCROLLBARS_AS_NEEDED. При этом полосы прокрутки видны, если размер содержимого панели превышает установленные размеры scrollpane.

Обработка событий

Все графические приложения работают по принципу обработки событий. События создаются операционной системой, Обычно события связаны с нажатиями на кнопки мыши или клавиши, но они могут генерироваться и процессором.

В этой лекции мы расскажем, как Java-программы оповещаются о событиях и как надо писать программы, чтобы они реагировали на события. Кроме того, мы рассмотрим использование специальных интерфейсов, позволяющих упростить написание программ, реагирующих на события.

Программирование событий в Java

Графический интерфейс всегда связан с генерацией событий. Каждое движение мыши, нажатие клавиши, изменение размера окна, перемещение окна и др. Генерируют события, которые приложение может обрабатывать. Обратите внимание на слово «может», именно это слово содержит суть обработки событий в современном программировании.

Любое приложение на Java или javabean обрабатывает только небольшое количество событий, которые передаются в них JVM. Если приложение реагировало бы на все события, то оно было бы огромных размеров и содержащим код для обработки каждого события.

Sun изменила ранее существующую обработку событий, ввела новую улучшенную стратегию обработки событий, основанную на использовании «прослушивателей событий» (event listeners). Прослушиватель событий – это класс, который реализует интерфейс java.awt.event.eventlistener, обычно через его наследников. Некоторые из таких интерфейсов приведены ниже:
Actionlistener, adjustmentlistener, changelistener, componentlistener, connectioneventlistener, containerlistener, controllereventlistener, focuslistener, itemlistener, keylistener, linelistener, listdatalistener, listselectionlistener,menulistener, mouseinputlistener, mouselistener, mousemotionlistener, mousewheellistener, windowstatelistener

Каждый из этих интерфейсов требует, чтобы один или несколько методов были определены в классе, который будет реализовать эти интерфейсы. Например, интерфейс mouselistener требует определения пяти методов:

- Mouseclicked(mouseevent e) ,
- Mouseentered(mouseevent e) ,
- Mouseexited(mouseevent e) ,
- Mousepressed(mouseevent e),

– Mousereleased(mouseevent e) .

Интерфейс actionlistener требует только один метод:

actionperformed(actionevent e)

Заметим, что в каждый метод передается параметр – некоторый объект класса с именем Event. Все эти классы наследуют от java.awt.Event. Каждый из этих классов содержит методы, специфичные для обслуживания события.Например, класс actionevent содержит четыре метода:

– Getactioncommand() возвращает командную строку, связанную с указанным действием.
– Getmodifiers() возвращает код модификатора, который был нажат при выработке события (Ctrl илиshift).
– Getwhen() возвращает время свершения события.
– Paramstring() возвращает строку-параметр для события.

Итак, JVM создает объект типа Event из того события, которе она получила от операционной системы. Созданный объект может использоваться при программировании для обработки.

Листинг показывает, как все это работает:

```
Import javax.swing.*;  import
java.awt.event.actionlistener;
import
javax.swing.border.etchedborder;
import java.awt.Container;
Import java.awt.borderlayout;

Import java.util.*;  public class
eventcreator extends jframe
{
   Jbutton btnbook;
   Jbutton btnexit;

   /** Constructor */
   Public eventcreator() throws Exception

{
tr
y
      {
```

```
        //configure the Frame
        Eventconsumer ec = new eventconsumer();
        Setbounds(150,200,500,250);
setdefaultcloseoperation(jframe.EXIT_ON_CLOSE);

        //set the layout
        Borderlayout border = new
borderlayout();          Container content =
getcontentpane();
content.setlayout(border);          btnbook =
new jbutton("Book");          btnexit = new
jbutton("Exit");
btnbook.addactionlistener(ec);
btnexit.addactionlistener(ec);

        Jpanel bottompanel = new jpanel();
bottompanel.add(btnbook);
bottompanel.add(btnexit);
content.add(bottompanel, borderlayout.SOUTH);
setvisible(true);       }catch(Exception e)
      {
        System.out.println("Exception thrown " + e);
      }
   }

   /**
    * @param args the command line arguments
    */
   Public static void main(String args[])
   {
     //create an instance of
the GUI       try       {
        Eventcreator mainwindow = new eventcreator();
      }catch(Exception e)
      {
        System.out.println("Exception in main " + e);
      }
   }
```

}//class

Здесь создаются две ссылки на кнопки типа jbuttons.

```
Jbutton btnbook;
Jbutton btnexit;
```

Затем мы создаем объект типа прослушивателя события. Прослушиватель события должен для кнопки реализовывать интерфейс actionlistener:

```
Eventconsumer ec = new eventconsumer();
```

Теперь мы создаем кнопки и добавляем с помощью метода addactionlistener() объект типа прослушивателя. Когда мы нажмем на кнопку именно в этот объект, а конкретно в метод actionperformed будет передано управление.

```
Btnbook = new
jbutton("Book");  btnexit = new
jbutton("Exit");
btnbook.addactionlistener(ec);
btnexit.addactionlistener(ec);
```

Ниже приводится код этого класса:

```
Import javax.swing.*;
Import java.awt.event.actionlistener;
import
javax.swing.border.etchedborder;
import java.awt.Container;
Import java.awt.borderlayout;

Import java.util.*;

Public class eventconsumer implements actionlistener
{
   Jbutton btnbook;
   Jbutton btnexit;
   /** Constructors for eventconsumer */
   Public eventconsumer() throws Exception
   {
   }
   Public void actionperformed(java.awt.event.actionevent ae)
   {
```

```
      If (ae.getactioncommand().equals("Exit"))
      {
         System.out.println("Exit was clicked");
      }
      //Try and book a ticket
      If (ae.getactioncommand().equals("Book"))
      {
         System.out.println("Book was clicked");
      }
   }
}//class
```

При запуске программы в консольном окне будут выводиться строки:

```
Book was clicked
Exit was clicked
Book was clicked
Exit was clicked
```

В этом примере показана основные правила, которые надо соблюдать, чтобы связать события, генерируемые каким-то компонентом, с кодом обработки события, написанным специально для реакции на это событие. Различные типы событий и различные методы для управления событиями существуют в Java. Но все они следуют одному образцу: прослуштватель события регистрируется на объекте, вырабатывающем событие.

Второй способ организации прослушивания события – создать единственный класс, который сам реализует интерфейс прослушивателя. Все, что нужно сделать – это реализовать в классе интерфейс actionlistener, а затем вызвать метод addactionlistener(), передавая в качестве параметра в него this, определяя, что этот класс сам обрабатывает событие. Листинг показывает этот пример:

```
Import javax.swing.*;  import
java.awt.event.actionlistener;
import
javax.swing.border.etchedborder;
import java.awt.Container;
Import java.awt.borderlayout;

Import java.util.*;

Public class eventcreator2 extends jframe implements actionlistener
```

```
{
   Jbutton btnbook;
   Jbutton btnexit;

   /** Constructors for cruiselist */
   Public eventcreator2() throws Exception

{
tr
y
      {
         //configure the Frame
         System.out.println("eventcreator2 executing");
setbounds(150,200,500,250);
setdefaultcloseoperation(jframe.EXIT_ON_CLOSE);

         //set the layout
         Borderlayout border = new
borderlayout();            Container content =
getcontentpane();
content.setlayout(border);            btnbook =
new jbutton("Book");            btnexit = new
jbutton("Exit");
btnbook.addactionlistener(this);
btnexit.addactionlistener(this);

         Jpanel bottompanel = new jpanel();
         Bottompanel.add(btnbook);
bottompanel.add(btnexit);
content.add(bottompanel, borderlayout.SOUTH);
setvisible(true);         }catch(Exception e)
      {
         System.out.println("Exception thrown " + e);
      }
   }
   Public void actionperformed(java.awt.event.actionevent ae)
   {
      If (ae.getactioncommand().equals("Exit"))
      {
```

```
        System.out.println("Exit was clicked2");
      }
      //Try and book a ticket
      If (ae.getactioncommand().equals("Book"))
      {
        System.out.println("Book was clicked2");
      }
   }
   /**
    * @param args the command line arguments
    */
   Public static void main(String args[])
   {
      //create an instance of
the GUI        try        {
        Eventcreator2 mainwindow = new eventcreator2();
      }catch(Exception e)
      {
        System.out.println("Exception in main " + e);
      }
   }
}//class
```

Теперь и генерация событий, и обработка их выполняются в одном классе. Объявление класса показывает, что он реализует дополнительный интерфейс: public class eventcreator2 extends jframe implements actionlistener

Вместо использования другого класса как прослушивателя actionlistener, теперь используется этот же класс:

```
Btnbook.addactionlistener(this);
btnexit.addactionlistener(this);  Метод actionperformed()
```

реализован в этом же классе:

```
Public void actionperformed(java.awt.event.actionevent ae)
{
   If (ae.getactioncommand().equals("Exit"))
   {
      System.out.println("Exit was clicked2");
   }
```

```
   //Try and book a ticket
   If (ae.getactioncommand().equals("Book"))
   {
      System.out.println("Book was clicked2");
   }

}
```

На практике можно использовать оба подхода: делать прослушиватель в отдельном классе или объединять все в одном классе. Теперь, когда мы рассказали, как выполнгяется обработка событий в Java, можно подробнее рассмотреть интерфейсы прослушивания.

Использование интерфейсов прослушивания

Достаточно большое количество интерфейсов поддерживается в Java. В этом разделе мы опишем самые распространенные.

Выбрасываемые события можно разделить на два вида: семантические или высокоуровневые события и низкоуровневые события. Высокоуровневые события связаны не с простым сигналом от внешнего устройства.

Высокоуровневые события в Java – это:

– Actionevent— нажата кнопка мыши или выбран пункт меню.
– Adjustmentevent— выполнено передвижение по строке прокрутки.
– Itemevent— пользователь сделал выбор из списка. – textevent— текстовое поле изменено.

Перечень низкоуровневых событий:

– Componentevent— был изменен компонент (передвинут, изменился размер, стал видимым.невидимым.
– Keyevent— была нажата или отпущена клавиша.
– Mouseevent— кнопка мыши была нажата или отпущена, мышь была передвинута или отпущена.
– Focusevent— компонент получил или потерял фокус.
– Windowevent— окно было свернуто, активировано, деактивировано и т. Д.
– Containerevent— компонент был помещен или удален из контейнера.

– Paintevent— это событие не используется прослушивателями, но содержит методы, которые можно перекрывать.
– Inputevent— это событие используется, чтобы не дать событию быть выброшенным компонентом.

Связанный с каждым событием класс – это прослушиватель-интерфейс с одним, по крайней мере, методом. Заметим, что событие mouseevent имеет два прослушивателя. Причина этого – то , что мышь передвигается по плоскости и имеет клавиши. Каждый из интерфейсов имеет один или более методов, связанных с ними. Интерфейс windowlistener имеет больше методов - семь. Интерфейсы actionlistener, adjustmentlistener, itemlistener и textlistener имеют только по одному методу.

Некоторые прослушиватели с несколькими методами имеют специальные классы, называемые адаптерами. Эти классы реализуют интерфейс и обеспечивают «пустые» методы. Программист должен наследовать от этих классов и перекрыть только те методы, которые необходимы. Если какие-то методы не реализованы, то ошибки не выбрасываются, потому что в адаптере методы реализованы.классы-адаптеры. В последующих примерах показано использование некоторых интерфейсов.

Интерфейс actionlistener

Интерфейс actionlistener – самый простой. В нем только один метод - actionperformed(). Этот метод получает событие типа java.awt.actionevent как параметр. Класс actionevent имеет три метода, которые можно использовать для определения уточнения некоторых условий, при которых произошло событие:

– Getactioncommand()— метод возвращает имя, связанное с командой, которая вызвала событие. Испльзуя этот метод, можно определить, какая кнопка или команда вызвала событие.
– Getmodifiers()— этот метод позволит определить, нажаты ли клавиши Shift или Ctrl, когда вызывалась команда или нажималась кнопка.
– Paramstring()— этот метод возвращает строку, связанную с событием, если она есть.

Список использованной и рекомендуемой литературы

1. Мейер Б. Объектно-ориентированное конструирование и программирование систем / Пер. С англ. – М: «Русская Редакция», 2005.
2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд.: Пер. С англ. – М.: Издательский дом “Вильямс”, 2008. – 720 с.
3. Мартин, Р. Быстрая разработка программ: принципы, примеры, практика.: Пер. С англ. – М.: Издательский дом «Вильямс», 2004. – 752 с.
4. Амблер, С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. – спб.: Питер, 2005. – 412 с.
5. Фаулер, М. Архитектура корпоративных программных приложений.: Пер. С англ. – М.: Издательский дом “Вильямс”, 2004. – 544 с.
6. Вигерс К., Битти Д. Разработка требований к программному обеспечению – 3-е изд, доп. - Пер. С англ. - М.: Русская редакция; спб.: БХВ-Петербург, 2018. – 736 стр.: ил.
7. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. Спб: Символ-Плюс, 2006. – 304 с., ил.
8. Дюваль М., Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска / Дюваль, М. Поль. – М,: Вильямс, 2008. – 240 с.
9. Гецци К., Джазаейри М., Мандриолн Д. Основы инженерии программного обеспечения. 2-е изд.: Пер. С англ. – спб.: БХВ-Петербург, 2005. – 832 с.: ил.
10. Благодатских В.А. Стандартизация разработки программных средств: учеб. Пособие /В.А. Благодатских, В.А. Волнин, К.Ф. Поскакалов; под ред. О.С. Разумова. - М.: Финансы и статистика, 2006. - 288 с; ил.
11. Котляров, В.П. Основы тестирования программного обеспечения / В.П. Котляров. – М.: Интернст-Университет Информационных Технологий (ИНТУИТ), 2015. – 147 с.
12. Орлов С. Технологии разработки программного обеспечения. Разработка сложных программных систем. Учебное пособие. Спб: Питер, 2003. 480 с, ил.
13. Таейр, Т. Надежность программного обеспечения / Т. Таейр, М. Липов, Э. Нельсое. – М.: Мир, 1981. – 323 с.

Оглавление

```
}
```