МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Гомельский государственный технический университет имени П.О. Сухого»

Кафедра «Информационные технологии»

Практические занятия
старшего преподавателя
Стефановского И.Л.
по дисциплине

«ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

для специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)»

Гомель 2024

Библиотека ГГТУ им.П.О.Сухого

Содержание

ВВЕДЕНИЕ 3
Практическая работа №1 «Жизненный цикл разработки программного обеспечения: этапы, модели и методологии» 4
Задание на практическую работу 27
Практическая работа №2 «Развитие представлений о разработке программ: от спагетти-кода к методологии структурного программирования. Объектно-ориентированное программирование в Java» 32
Задание на практическую работу 36
Практическая работа №3 «Инструменты программирования и отладки: интегрированная среда разработки, система контроля версий, системы управления проектами» 40
Задание на практическую работу 61

Библиотека ГГТУ им.П.О.Сухого

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

Библиотека ГГТУ www.no.Cyroro

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».

Практическая работа №1 «Жизненный цикл разработки программного обеспечения: этапы, модели и методологии»

Цель работы: изучить жизненный цикл разработки программного обеспечения

Теоретические сведения

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

Библиотека ГГТУ им. П.О.Сухого

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,

Библиотека ГГТУ им. П.О.Сухого

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

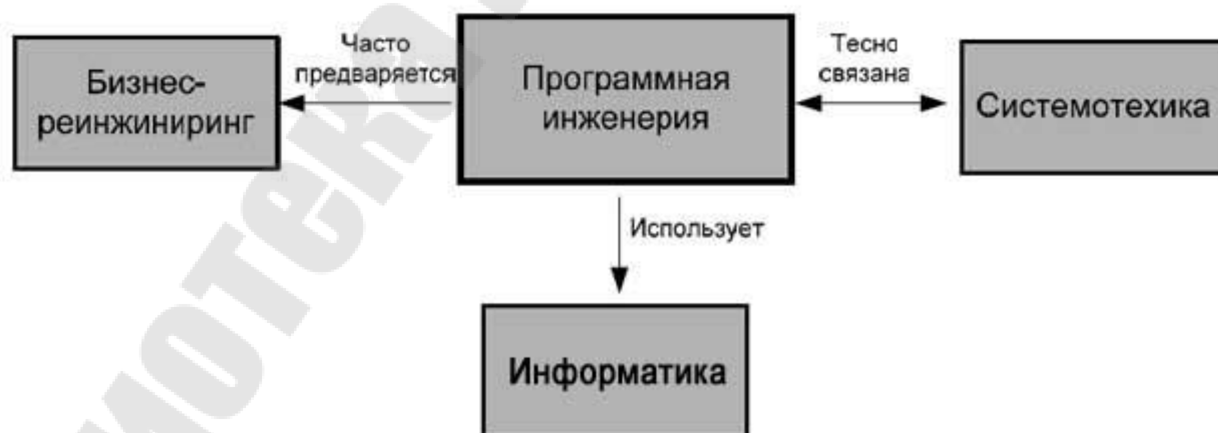

Рис. 1.1. Взаимосвязи программной инженерии

Программное обеспечение

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

Библиотека ГГТУ им. П.О. Сухого

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

Библиотека ГГТУ им. П.О. Сухого

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

Библиотека ГГТУ им.П.О.Сухого

(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*.

Библиотека ГГТУ им. П.О.Сухого

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*

Библиотека ГГТУ им. П.О. Сухого

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 (разработчик ИС)	– Инициирование – Ответ на заявочные предложения – Подготовка договора – Планирование исполнения – Поставка ИС	– Техническое задание на ИС – Решение руководства об участии в разработке – Результаты тендера – Техническое задание на ИС – План управления проектом – Разработанная ИС и документация	– Решение об участии в разработке – Коммерческие предложения/ конкурсная заявка – Договор на поставку/ разработку – План управления проектом – Реализация/ корректировка

Библиотека ГГТУ им. П.О. Сухого

			– 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 тестирование ИС	– 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, документация на ИС, тесты	– 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 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

Библиотека ГГТУ им. П.О. Сухого

			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;

Библиотека ГГТУ им. П.О. Сухого

– 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).

Библиотека ГГТУ им. П.О.Сухого

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,

Библиотека ГГТУ им. П.О.Сухого

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).

Библиотека ГГТУ им. П.О.Сухого

Рис. 1.2. Каскадная модель ЖЦ ИС

Рис. 1.3. Поэтапная модель с промежуточным контролем

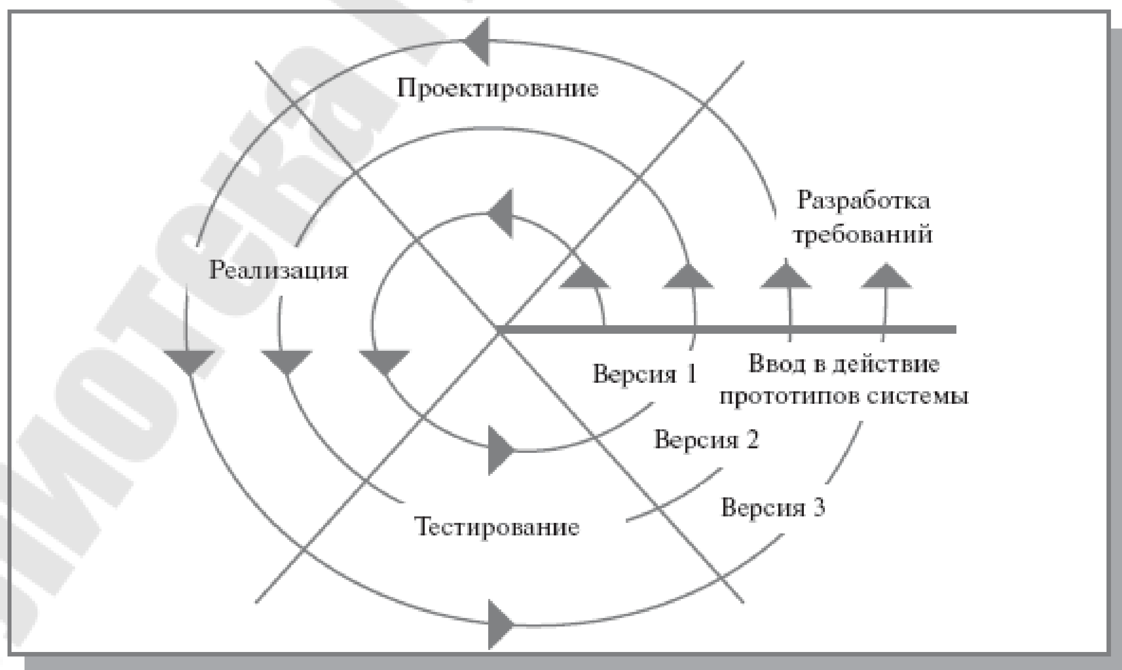

Рис. 1.4. Спиральная модель ЖЦ ИС

Библиотека ГГТУ им. П.О.Сухого

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

Библиотека ГГТУ им. П.О.Сухого

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

Библиотека ГГТУ им. П.О.Сухого

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.)

Библиотека ГГТУ им. П.О. Сухого

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

Библиотека БГУИР

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

Библиотека ГГТУ им. П.О.Сухого

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.

Библиотека ГГТУ им. П.О. Сухого

– 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).

Библиотека ГГТУ им. П.О.Сухого

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*

Библиотека БГУИР www.bsuir.by

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.

Библиотека ГГТУ им.П.О.Сухого

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

Задание на практическую работу

1. Описать жизненный цикл разработки программного обеспечения согласно варианта (таблица 1) используя каскадную, поэтапную и спиральную модели.
2. Разработать *UML*-диаграмму иерархии классов, согласно варианта (таблица 1).
3. При наименовании компонентов руководствоваться соглашением о наименовании (https://www.oracle.com/technetwork/java/codeconventions-150003.pdf).
4. При описании иерархии использовать наследование и композицию.
5. На основе *UML*-диаграммы разработать иерархию классов на языке *Java*.
6. Весь код должен быть снабжен элементами документирования (https://www.jetbrains.com/help/idea/working-with-code-documentation.html).
7. Разработанную иерархию поместить в *.jar* файл для дальнейшего использования в качестве библиотечных классов.

Библиотека ГГТУ им. П.О. Сухого

8. Создать консольное приложение для демонстрации работы созданных классов.
9. Составить отчет о проделанной работе.

Таблица 1

Вариант	**Условие задачи**
1	1.1 Создать иерархию классов для учёта покупок хот-догов (*HunterDog*, *MasterDog* и *Berlinka*); 1.2 При этом все компоненты могут продаваться отдельно. 1.3 Продать каждый вид хот-дога. 1.4 Подсчитать общую сумму всех заказов. 1.5 Подсчитать количество полных заказов. 1.6 Подсчитать среднюю стоимость заказов.
2	1.1 Разработать иерархию классов для хранения информации о деталях: форма, материал, вес, размер. 1.2 Создать не менее 10 объектов, содержащих информацию о различных деталях. 1.3 Подсчитать общий вес деталей, содержащие одинаковую форму. 1.4 Вывести количество деталей. 1.5 Вывести деталь, которая отличается своей формой.
3	1.1 Создать класс *Sneakers* (Кроссовки) с полями *name*, *cost*, *type*, *producer*. Кроссовки могут быть футбольные, баскетбольные и т.п. (реализовать через наследование) 1.2 Создать приватный внутренний класс *Producer* с полями *name* и *country*.
	1.3 Создать не менее 10 объектов, содержащих информацию о различных кроссовках 1.4 Подсчитать количество производителей. 1.5 Подсчитать среднюю стоимость обуви по каждому производителю. 1.6 Подсчитать среднюю стоимость обуви по каждому типу. 1.7 Дополнить иерархию классов новым типом кроссовок, при этом нельзя изменять методы, реализующие пункты 1.4 – 1.6

Библиотека ГГТУ им. П.О.Сухого

4	1.1 Создать иерархию классов для учёта выплаты стипендии студентам по итогам экзаменационной сессии. 1.2 Только студенты, обучающиеся на бюджетной форме и сдавшие сессию в срок, получают стипендию. 1.3 Студент, имеющий средний бал меньше 5 стипендию не получает, от 6 до 7 получают минимальную стипендию, от 6 до 8 – увеличенную на 25%, от 8 до 10 – на 50%. 1.4 Ввести данные с информацией о 10 студентах, которые получили стипендию, увеличенную на 25%. 1.5 Вывести студентов, обучающихся на платной основе. 1.6 Вывести средний балл по итогам сессии по всем студентам.
5	1.1 Создать класс *Car*, *Engine* и *Driver*. 1.2 Класс *Driver* содержит поля - ФИО, стаж вождения. 1.3 Класс *Engine* содержит поля - мощность, производитель. 1.4 Класс *Car* содержит поля - марка автомобиля, класс автомобиля, вес, водитель типа *Driver*, мотор типа *Engine*. 1.5 Вывести водителей со стажем более 5 лет. 1.6 Вывести автомобили советского производства.
6	1.1 Создать объект класса Компьютер, используя классы Винчестер, Дисковод, ОЗУ. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера. 1.2 Создать не менее 10 объектов. 1.3 Добавить возможность докупать компоненты. 1.4 Вывести всю информацию о компьютере. 1.5 Вывести компьютеры, которые собирались вручную. 1.6 Подсчитать самый выгодный компьютер.
7	1.1 Создать иерархию классов для учёта самолетов в аэропорту. 1.2 Создать 3 класса самолетов и минимум 3 экземпляра. 1.3 Классы должны содержать поля: количество пассажиров, количество топлива, название рейса. 1.4 Рассчитать расход топлива.
	1.5 Вывести все самолеты, которые содержать количество пассажиров меньше заданного. 1.6 Вывести все самолеты, название которых начинается с букву заданную букву.

Библиотека ГГТУ им. П.О. Сухого

8	1.1 Создать иерархию для учета книг в библиотеках. 1.2 Библиотека содержит в себе название и коллекцию книг, а также методы добавления книги. 1.3 Класс книги содержит поля: название, автор, год написания. 1.4 Создать несколько объектов библиотек и добавить книги в каждую. 1.5 Вывести все книги одного автора из каждой библиотеки. 1.6 Вывести всю информация по книге(название, автор, год написания).
9	1.1 Создать иерархию для учета блюд в меню. 1.2 Меню содержит блюда: напитки и тосты (реализовать через наследование). Напитки и тосты имеют название, стоимость, калорийность и метку *vegan friendly*. 1.3 Создать объект меню и не менее 10 различных блюд. 1.4 Вывести все меню. 1.5 Вывести все *vegan friendly* блюда. 1.6 Подсчитать среднюю стоимость блюд с калорийностью более 300.
10	1.1 Создать иерархию комнатных растений с полями название, высота(размер), продолжительность жизни. 1.2 Растения могут быть нескольких видов: Красивоцветущие, Суккуленты, Папоротники. Реализовать через наследование. У красивоцветущих добавить поле продолжительность цветения. 1.3 Создать не менее 10 объектов различных растений. 1.4 Вывести всю информацию по красивоцветущим. 1.5 Вывести все растения, продолжительность жизни которых менее 5 лет. 1.6 Вывести все папоротники, высота которых больше заданной.
11	1.1 Создать абстрактный класс пицца и наследовать от него виды Пепперони, Сырная, Мясная. 1.2 Каждая пицца содержит поля цена, вес, диаметр и калорийность. 1.3 Создать не менее 3 экземпляров каждого класса пицц. 1.4 Вывести количество пицц, вес которых превышает заданный. 1.5 Вывести цену каждой пиццы, диаметр которой превышает заданный.

Библиотека ГГТУ им. П.О. Сухого

12	1.1 Создать абстрактный класс Булка и наследовать от него минимум 3 вида булок. 1.2 Абстрактный классный должен содержать поля: цена, ингредиенты, дата момента создания. 1.3 Создать не менее 10 объектов различных булок с разным промежутком выпечки. 1.4 При этом ингредиенты можно добавить отдельно. 1.4 Вывести все булки, которые были созданы 5 мин. назад. 1.5 Вывести булки в которых есть перец.
13	1.1 Создать иерархию Аниме, три типа: сериал *ONA*, *OVA*. 1.2 Класс Аниме должен содержать поля: жанры(коллекция), рейтинг, количество серий, дата выхода. 1.3 Создать не менее 10 экземпляров. 1.4 Вывести все Аниме, рейтинг которых больше 7. 1.5 Вывести Аниме, которые вышли недавно. 1.6 Перечислить жанры Аниме, которые встречаются чаще всего.
14	1.1 Создать иерархию классов для учета рыб в Аквариуме. 1.2 В классе рыба реализовать поля: вид (хищная, мирная и т.д.), вес, размер (отдельный класс с полями ширина и длина). 1.3 Создать не менее 5 видов рыб, каждой по 2 экземпляра. 1.4 Вывести информацию о видах рыб, которые являются хищниками и весят больше 100 гр. 1.5 Вычислить самую длинную рыбу.
15	1.1 Создать абстрактный класс *Food* и наследовать от него фрукты, овощи, мясо. 1.2 Создать в классе *Food* метод расчета энергетической ценности 1.3 В классе *Food* создать поля: съедобное / несъедобное, коллекция пищевой ценности (белки, жиры, углеводы), название. 1.4 Создать не менее 10 объектов. 1.5 Вывести все съедобные фрукты, пищевая ценность которых меньше заданного числа. 1.6 Вывести еду, название которых заканчивается на введенную букву.

Контрольные вопросы

Библиотека ГГТУ им. П.О. Сухого

1. Что такое сигнатура метода? А контракт метода?
2. Что такое интерфейс?
3. Что можно сделать при помощи переменной super?
4. В чем отличие результата применения модификатора final к классу и к переменной?
5. Что такое абстрактный метод?
6. Будет ли ошибка выдана компилятором, если класс реализует два интерфейса, имеющих методы с одинаковой сигнатурой?
7. Можно ли объявить двумерный массив так:
   int[] twoDimArray[] = new int[10][];?
8. Чем абстрактный класс отличается от интерфейса?
9. Что произойдет с переменной nameOfChild в результате выполнения операторов:

```
    String nameOfChild = "Bill";
nameOfChild += " Gates";
```

10. Что делает конструктор класса? Должен ли он обязательно явно присутствовать в объявлении класса?
11. Какие в существуют виды ссылочных типов? Как реализуются ссылочные переменные?

Практическая работа №2 «Развитие представлений о разработке программ: от спагетти-кода к методологии структурного программирования. Объектно-ориентированное программирование в Java»

Цель работы: изучить объектно-ориентированное программирование в Java

Теоретические сведения

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.

Библиотека ГГТУ им. П.О. Сухого

```
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
```

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.

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.

Библиотека ГГТУ им. П.О.Сухого

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.

Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора import такова:

import пакет1 [.пакет2].(имякласса|);*

Здесь *пакет1* — имя пакета верхнего уровня, *пакет2* — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import : *import java.util.Date*

import java.io.;*

Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за

Библиотека БГТУ им. П.О. Сухого

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).

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.

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.

Библиотека ГГТУ им. П.О. Сухого

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);
}

Задание на практическую работу

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **set**Тип(), **get**Тип(), **toString**(). Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

Варианты:

1. **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Создать массив объектов. Вывести:
a) список студентов заданного факультета;
b) списки студентов для каждого факультета и курса;
c) список студентов, родившихся после заданного года;
d) список учебной группы.

2. **Customer**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета. Создать массив объектов.
Вывести:
a) список покупателей в алфавитном порядке;
b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. **Patient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз. Создать массив объектов.
Вывести:
a) список пациентов, имеющих данный диагноз;

Библиотека БГТУ им. Сухого

b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки. Создать массив объектов. Вывести:
a) список абитуриентов, имеющих неудовлетворительные оценки;
b) список абитуриентов, средний балл у которых выше заданного;
c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет. Создать массив объектов. Вывести: a) список книг заданного автора;
b) список книг, выпущенных заданным издательством;
c) список книг, выпущенных после заданного года.

6. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации. Создать массив объектов. Вывести:
a) список квартир, имеющих заданное число комнат;
b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
c) список квартир, имеющих площадь, превосходящую заданную.

7. **Phone**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров. Создать массив объектов. Вывести:
a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;
b) сведения об абонентах, которые пользовались междугородной связью;
c) сведения об абонентах в алфавитном порядке.

8. **Car**: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер. Создать массив объектов. Вывести:
a) список автомобилей заданной марки;
b) список автомобилей заданной модели, которые эксплуатируются больше n лет;
c) список автомобилей заданного года выпуска, цена которых больше указанной.

9. **Product**: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.

Библиотека БГУИР www.nio.сухого

Создать массив объектов. Вывести:
a) список товаров для заданного наименования;
b) список товаров для заданного наименования, цена которых не превосходит заданную;
c) список товаров, срок хранения которых больше заданного.

10. **Train**: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс). Создать массив объектов.
Вывести:
a) список поездов, следующих до заданного пункта назначения;
b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;
c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. **Bus**: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег. Создать массив объектов.
Вывести:
a) список автобусов для заданного номера маршрута;
b) список автобусов, которые эксплуатируются больше 10 лет;
c) список автобусов, пробег у которых больше 100000 км.

12. **Airlines**: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.
Создать массив объектов. Вывести:
a) список рейсов для заданного пункта назначения;
b) список рейсов для заданного дня недели;
c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

13. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.
Создать массив объектов. Вывести:
a) список абитуриентов, имеющих неудовлетворительные оценки;
b) список абитуриентов, средний балл у которых выше заданного;
c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

14. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.
Создать массив объектов. Вывести: a)
список книг заданного автора;
b) список книг, выпущенных заданным издательством;

Библиотека ГГТУ им. П.О. Сухого

c) список книг, выпущенных после заданного года.

15. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации. Создать массив объектов. Вывести:
a) список квартир, имеющих заданное число комнат;
b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
c) список квартир, имеющих площадь, превосходящую заданную.

Контрольные вопросы

1. Правилен ли оператор определения переменной: double myValue = 1f; 2. Чем оператор break в языке Java отличается от оператора break в C/C++?
3. Как виртуальная машина Java (интерпретатор) вычисляет значения выраажений?
4. Что такое отрицательная бесконечность?
5. Чем различаются строковые литералы и переменные типов String, StringBuffer и StringBuilder?
6. Какие исключения могут быть возбуждены при арифметической обработке данных?
7. Каким ограничениям должно удовлетворять выражение в предложении case переключателя?
8. Охарактеризуйте назначение составных частей оператора перехвата исключений.
9. Где ошибка в этом операторе:
   switch (индексСимвола)
   { default : … break; case 5L: case 7: … break; … }
   10. Какие виды литералов существуют в языке
Java?
11. Правильно ли объявление: boolean flag = 0; ?

Библиотека ГГТУ им. П.О. Сухого

Практическая работа №3 «Инструменты программирования и отладки: интегрированная среда разработки, система контроля версий, системы управления проектами»

Цель работы: изучить инструменты программирования и отладки

Теоретические сведения

Система контроля версий

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.

Библиотека ГГТУ им. П.О. Сухого

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*/*libgit*2/*libgit*2

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*/*libgit*2/*libgit*2 *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.

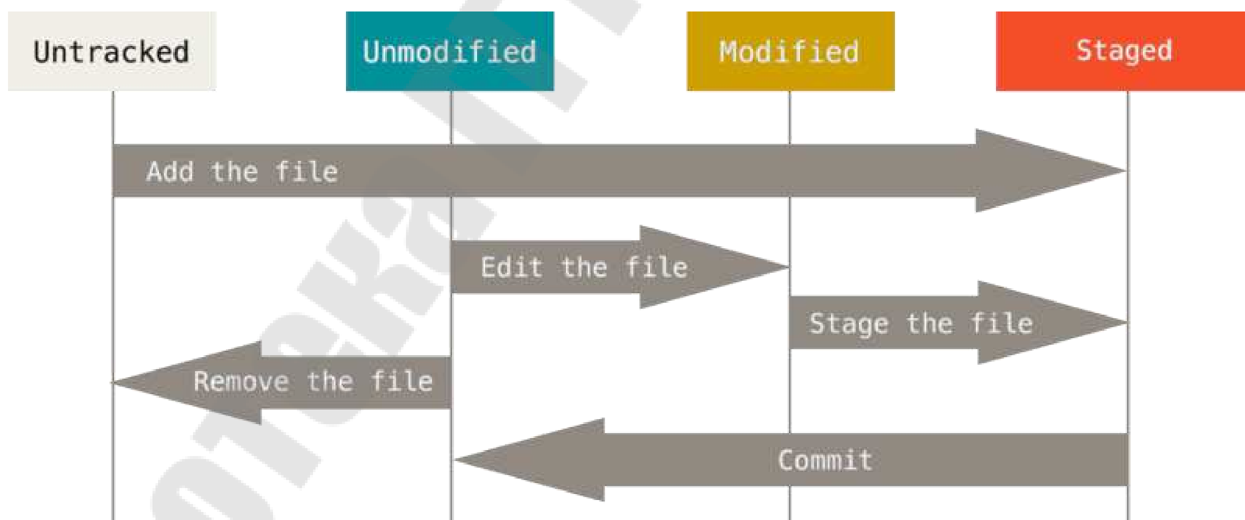

Рис. 4.1. Жизненный цикл состояний файлов

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

Библиотека БГТУ им. Шухова

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*

Библиотека ГГТУ им. П.О. Сухого

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
```

Библиотека БГТУ им. П.О. Сухого

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:

Библиотека ГГТУ им. П.О.Сухого

```
$ 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.

Библиотека ГГТУ им. П.О.Сухого

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
```

Библиотека ГГТУ им. П.О. Сухого

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 8ebb991..643e24f 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..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project

Библиотека ГГТУ им. П.О. Сухого

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/libgit*2*/libgit*2*/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;

Библиотека ГГТУ им. П.О. Сухого

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.

Библиотека ГГТУ им. П.О. Сухого

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 463*dc*4*f] 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

Библиотека ГГТУ им. П.О.Сухого

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

Библиотека ГГТУ им. П.О.Сухого

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
```

Библиотека ГГТУ им. П.О. Сухого

Bakkdoor https://github.com/bakkdoor/grit (fetch)
Bakkdoor https://github.com/bakkdoor/grit (push)
Cho45 https://github.com/cho45/grit (fetch)
Cho45 https://github.com/cho45/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*

Библиотека ГГТУ имени П.О.Сухого

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

Библиотека БГТУ им. П.О. Сухого

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
```

Библиотека ГГТУ им.П.О.Сухого

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**

Библиотека ГГТУ им.П.О.Сухого

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*

Библиотека БГУИР

*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
*AAAAB*3*NzaC*1*yc*2*EAAAABIwAAAQEAklOUpkDHrfHY*17*SbrmTIpNLTGK*9
Tjom/*BWDSU*
Gpl+*nafzlhdtyw*7*hdi*4*yz*5*ew*18*jh*4*jw*9*jbhufrviqzm*7*xlelevf*4*h*9*lfx*5*qvkbpp*
*pswg*0*cda*3
*Pbv*7*kOdJ*/*mtyblwxfcr*+*hao*3*fxritbqxix*1*nkhxphazsmcilq*8*v*6*rjsnaqwdsd*
mfvslvk/7*XA*

Библиотека ГГТУ им. П. О. Сухого

T3faojoasncm1q9x5+3V0Ww68/eifmb1zuufljqjkprrx88xypndvjynby6vw/Pb0rwert/En
Mz+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nkatmikjn2so1d01qratlmqvssbx
Nrrfi9wrf+M7Q== schacon@mylaptop.local

Задание на практическую работу

1. Создать локальный git репозиторий.
2. Создать три ветки (main, *lr*1, *lr*3). Закоммитить лабораторные работы 1 и 3 в ветки *lr*1 и *lr*3 соответственно.
3. Создать удаленный репозиторий на *GitHub*. Привязать локальный репозиторий к удаленному. Запушить ветки в удаленный репозиторий.
4. Создать и одобрить пул-реквесты на слияние веток *lr*1 и *lr*3 с веткой main.

Составить отчет о проделанной работе. Отчет должен содержать описание и скриншоты всех этапов выполнения лабораторной работы.

Контрольные вопросы

1. Работа с репозиториями.
2. Создание репозитория.
3. Аутентификация. SSH-ключ.
4. Клонирование репозитория.
5. Добавление удалённых репозиториев.
6. Синхронизация локального репозитория с удалённым.
7. Инспекция удалённого репозитория.
8. Добавление новых файлов или изменений в существующих.
9. Коммиты.
10. Обновление локальной копии репозитория.
11. Создание меток репозитория.

Библиотека ГГТУ им. П. О. Сухого