Министерство образования Республики Беларусь

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

Кафедра «Информационные технологии»

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ

Учебно-методическое пособие
для студентов специальностей 1-40 05 01
«Информационные системы и технологии
(по направлениям)» и 1-40 80 04 «Информатика
и технологии программирования»
дневной и заочной форм обучения

Гомель 2022

УДК 004.75(075.8)
ББК 32.973.4я73
Н53

Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 6 от 01.02.2021 г.*)

Учебно-методическое пособие выполнено в ходе реализации проекта MaCICT (Modernisation of Master Curriculum in Information Computer Technologies) в рамках Erasmus+ Программы Европейского Союза. В этом проекте представлены лучшие практики вузов-партнеров из Евросоюза для модернизации учебного плана и дисциплин с целью внедрения в учебный процесс изучения гибких навыков, позволяющих повысить конкурентоспособность на рынке труда выпускников ИТ-специальностей.

Составители: *И. Л. Стефановский*, *Д. В. Соболев*

Рецензент: зав. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Ю. В. Крышнёв*

Непрерывная интеграция : учеб.-метод. пособие для студентов
Н53 специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» днев. и заоч. форм обучения / сост.: И. Л. Стефановский, Д. В. Соболев. – Гомель : ГГТУ им. П. О. Сухого, 2022. – 197 с.

ISBN 978-985-535-491-9.

Знакомит студентов с использованием концепций непрерывной интеграции для информационных систем уровня предприятия, со средствами обработки больших объемов информации.

Для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения.

УДК 004.75(075.8)
ББК 32.973.4я73

ISBN 978-985-535-491-9

Оглавление

ПРЕДИСЛОВИЕ

Современный специалист в области разработки программного обеспечения должен уметь пользоваться технологиями непрерывной интеграции и владеть основами создания приложений различного уровня сложности.

Дисциплина «Непрерывная интеграция» знакомит студентов и магистрантов с основными концепциями, технологиями непрерывной интеграции для разработки информационных систем уровня предприятия.

Целью указанной дисциплины является подготовка специалиста, владеющего знаниями и практическими навыками по технологиям непрерывной интеграции информационных систем уровня предприятия.

К основным задачам дисциплины относится следующее:

– изучение студентами и магистрантами теоретических основ и технологий непрерывной интеграции информационных систем уровня предприятия;

– приобретение студентами и магистрантами практических навыков по технологиям непрерывной интеграции.

– освоение навыков работы с различными репозиториями; изучение фаз жизненного цикла создания программного продукта;

– изучение приложений по автоматизации доставки кода программного продукта из репозитория;

– изучение приложений по автоматизации сборок программного продукта;

– изучение приложений по автоматизации процесса тестирования программного продукта на различных этапах жизненного цикла;

– изученне методик контроля версионности сборок и идентификации рабочих продуктов;

– изучение методик стабилизации результатов работы;

– изучение методик отслеживания запросов на изменение;

– знакомство с нюансами параллельности разработки программного обеспечения.

В результате изучения дисциплины магистрант должен:

• знать:

– организационно-технические аспекты непрерывной интеграции;

– способы обеспечения непрерывной интеграции;

– основные методы непрерывной интеграции баз данных;

– основные методы непрерывного тестирования;

– основные технологии непрерывной инспекции;

– основные методы и технопогии непрерывного развертывания;

– основной инструментарий организации процесса непрерывной интеграции;

– базовые процедуры управления конфигурацией;

– виды репозиториев и инструменты работы с ними;

– фазы сборки программного продукта;

• уметь:

– работать с репозиториями и настраивать приложения для работы с ними;

– создавать код процесса развертывания и сценарии развертывания;

– работать с виртуальными машинами;

– настраивать и работать с межплатформенными системами сборки и управления проектами;

– вести автоматический контроль версий продукта;

– управлять процессом развертывания программного продукта;

– работать с различными модулями и расширениями инструментария непрерывной интеграции;

– настраивать инструмент развертывания и настройки тестовых стендов;

• владеть:

– исследовательскими навыками;

– навыками организации процесса непрерывной интеграции программного продукта.

Данное учебно-методическое пособие предназначено для обучения магистрантов на первой ступени получения высшего образования по специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» и на второй ступени получения высшего образования по специальности 1-40 80 04 «Информатика и технологии программирования».

ГЛАВА 1
ОСНОВНЫЕ ПОНЯТИЯ И ИНСТРУМЕНТЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Модели жизненного цикла программного обеспечения

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 – предварительный макет, эскиз, план системы на бумаге). Трудозатраты на анализ и проектирование, а также форма представления их результатов сильно варьируются в зависимости 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, т. е. оставляются многочисленные семантические следы, помогающие создать и сохранить, поддерживать единую, стройную архитектуру, единообразный стиль, порядок.

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

для возможности использовать разнообразный положительный опыт, достигнутый другими командами и разработчиками, этот самый опыт подвергается осмыслению, обобщению и надлежащему оформлению. Так, появляются различные методы и практики (*best practices*) – тестирования, проектирования, работы над требованиями и пр., архитектурных шаблонов и др., а также стандарты и методологии, касающиеся всего процесса в целом (например, *MSF*, *RUP*, *CMMI*, *Scrum*). Вот эти 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 гг. 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 г. Годом рождением программной инженерии является 1968 г. – 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 программного обеспечения (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 расскажем о взаимосвязях программной инженерии.

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

Системотехника (*system engineering*) объединяет различные инженерные дисциплины по разработке всевозможных искусственных систем – энергоустановок, телекоммуникационных систем, встроенных систем реального времени и т. д. Очень часто ПО оказывается частью таких систем, выполняя задачу управления соответствующего оборудования. Такие системы называются *программно-аппаратными*,

и участвуя в их создании, программисты вынуждены глубоко разбираться в особенностях соответствующей аппаратуры.

Бизнес-реинжиниринг (*business reengineering*) – в широком смысле обозначает модернизацию бизнеса в определенной компании, внедрение новых практик, поддерживаемых соответствующими новыми информационными системами (ИС). При этом акцент может быть сделан 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*** множество развивающихся во времени логических предписаний, с помощью которых определенный коллектив людей управляет многопроцессорной и распределенной системой вычислительных устройств и использует ее.

Определение, данное Харальдом Милсом, известным специалистом в области программной инженерии из компании *IBM* включает в себя следующее:

• Логические предписания – это не только сами программы, но и различная документация (например, по эксплуатации программ), а 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, серверы 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.

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

Свойства. Программное обеспечение является сложной динамической системой, включающей в себя технические, психологические и социальные аспекты. Оно заметно отличается от других видов систем, создаваемых (созданных) человеком, – механических, социальных, научных и др., 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 т. д.) 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 естественно-научное, 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 базируются на постулатах и аксиомах), а должно быть согласовано с большим количеством интерфейсов, с которыми впоследствии

ему необходимо взаимодействовать. Эти интерфейсы плохо поддаются стандартизации, поскольку строятся 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. Поэтому неслучайно, что процесс оказался одним из основных понятий программной инженерии. Центральным объектом изучения программной инженерии является процесс создания.

Программное обспечение – это 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 др.).

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

шаблон процесса, создаваемый или адаптируемый (в случае использования стандартного) перед началом разработки. В *VSTS* существуют заготовки для конкретных процессов на базе *CMMI*, *Scrum* и др.

В рамках компании возможны и полезны объединение и стандартизация всех текущих процессов, которые мы будем называть стандартным процессом. Последний, таким образом, оказывается некоторой базой данных, содержащей следующее:

– информацию, правила использования, документацию и инсталляционные пакеты средств разработки, используемые 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 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, случается, 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 офшорной компании позволяет ей брать более широкий спектр заказов). Но очень часто это произвольный выбор самих разработчиков.

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

Необходимо отметить, что наличие стандартного процесса свидетельствует о наличии «единой воли» в организации, существующей именно на уровне процесса. На уровне продаж, бухгалтерии и других 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 офшорных проектах) – «текучка» захватывает и изолирует проекты друг от друга очень прочно.

В соответствии с базовым международным стандартом *ISO*/*IEC* 12207 все процессы жизненного цикла программного обеспечения (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 табл. 1.1 приведены ориентировочные описания основных процессов ЖЦ. Вспомогательные процессы предназначены для поддержки выполнения основных процессов, обеспечения качества проекта, организации верификации, проверки и тестирования ПО. Организационные процессы определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами.

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

Таблица 1.1

Содержание основных процессов жизненного цикла программного обеспечения информационных систем (*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. • 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.	• Техническое задание на ИС. • Техническое задание на ИС, модель ЖЦ. • Подсистемы ИС. • Спецификации требования к компонентам ПО. • Архитектура ПО. • Материалы детального проектирования ПО.	• Используемая модель ЖЦ, стандарты разработки. • План работ. • Состав подсистем, компоненты оборудования. • Спецификации требования к компонентам ПО.

Окончание табл. 1.1

Процесс (исполнитель процесса)	**Действия**	**Вход**	**Результат**
	• Кодирование тестирование ПО. • Интеграция ПО и квалификационное тестирование ПО. • Интеграция ИС и квалификационное тестирование ИС	• План интеграции ПО, тесты. • Архитектура ИС, ПО, документация на ИС, тесты	• 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 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 др. 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 необходимо включать следующие группы процессов:

1. *Договорные процессы*:

– приобретение (внутренние решения или решения внешнего поставщика);

– поставка (внутренние решения или решения внешнего поставщика).

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

Одним из базовых понятий методологии проектирования ИС является понятие жизненного цикла ее программного обеспечения. Жизненный цикл 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:

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

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

3) 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, в том числе оформление проектной и эксплуатационной документации, подготовку материалов, применяемых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т. д. Разработка ПО состоит, как правило, из анализа, проектирования и реализации (программирование).

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

Управление проектом связано с вопросами планирования и организации работ, с созданием 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 исходным требованиям. Проверка частично

совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

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

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. Жизненный цикл 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 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.

Рис. 1.2. Каскадная модель ЖЦ ИС

• *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.

Рис. 1.3. Поэтапная модель с промежуточным контролем

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

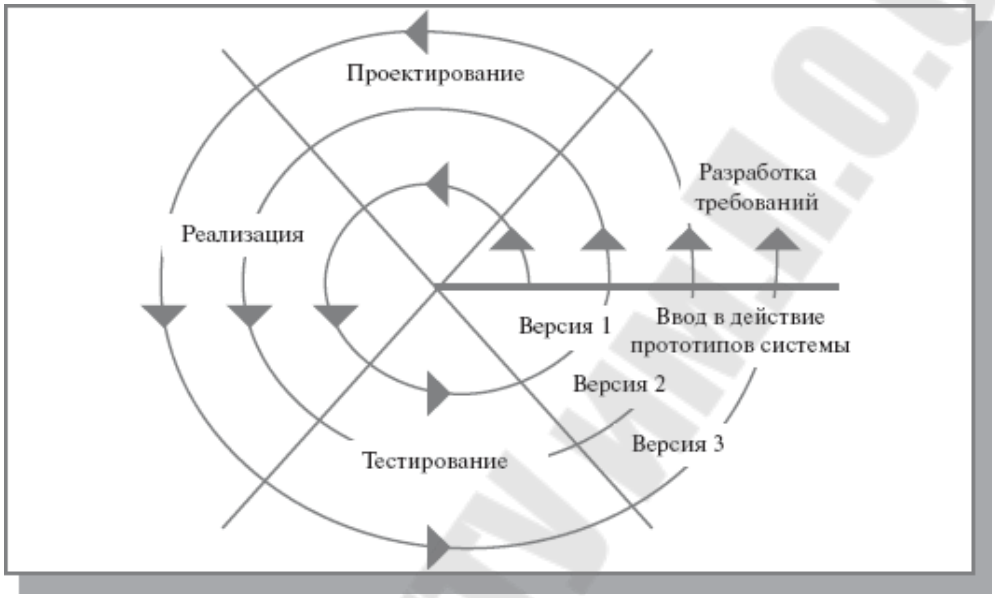

Рис. 1.4. Спиральная модель ЖЦ ИС

На практике наибольшее распространение получили две основные модели жизненного цикла:

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

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

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

Можно выделить следующие положительные стороны применения каскадного подхода:

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

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

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

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

Спиральная модель. Процесс итеративной (или инкрементальной) разработки стал эволюционным развитием модели водопада. Жизненный цикл был предложен для преодоления перечисленных проблем. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

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

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланиро-

ванная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

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

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

2. *Иллюзия снижения рисков* участников проекта (заказчика и исполнителя). Каскадная модель предполагает разработку законченных продуктов на каждом этапе: технического задания, технического проекта, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и (или) противоречивы), то в действительности использование каскадной модели создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки (*fixed price*). Второй тип предполагает повременную оплату работы (*time work*). Выбор того или иного типа контракта зависит от степени определенности задачи. Каскадная модель с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых

затрат, поэтому используются контракты с фиксированной ценой и, следовательно – каскадная модель разработки и внедрения. Спиральная модель чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.

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

Гибкие методологии

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

В 2001 г. 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 наиболее 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 г. в ходе попытки спасти провальный проект по разработке системы расчета заработной платы для компании «Крайслер». В 2000 г. проект был закрыт, но *XP* к тому времени уже получила известность и начала распространяться среди разработчиков ПО.

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

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

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

– разработка через тестирование (*test driven development*) – в *XP* становится особенно важным, чтобы весь создаваемый код был покрыт автоматическими юнит-тестами (почему это так, станет понятно дальше). Этого можно добиться при помощи простого правила – новый код может быть написан исключительно для того, чтобы уве-

личить число успешно проходящих юнит-тестов. Фактически это означает, что перед реализацией новой функции разработчик должен создать соответствующий тест, а написание кода должно завершиться в тот момент, когда начнет проходить новый, а также все существующие тесты. Если после этого окажется, что новая функция реализована не полностью, необходимо создать еще один тест и повторить весь цикл заново;

– архитектура системы должна быть максимально простой. *XP* не рекомендует проектировать в расчете на будущее развитие системы; идеальная архитектура должна не более чем поддерживать существующую функциональность. Цель такого минималистичного подхода к проектированию – избежать бесполезных инвестиций в архитектурные решения, которые часто оказываются выброшенными после очередного изменения требований. Вместо этого архитектура постоянно изменяется и развивается вместе с системой;

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

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

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

– продолжительность рабочей недели не должна превышать 40 ч. 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:

1. 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.

2. Истории сортируются по приоритету, рискам, сложности реализации.

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

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

5. Итерация планируется аналогичным образом.

6. Предварительно определяется набор историй для итерации.

7. 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».

8. 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.

9. 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.

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

Практика

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

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

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

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

Практика поддержания максимально простой архитектуры может завести в тупик в конце проекта, когда окажется, что для реализации завершающих историй требуется полное перепроектирование системы (оказывается, нам нужно было предусмотреть возможность

интеграции с системой *SAP R*/3, а также перевода на японский язык всего пользовательского интерфейса!). Даже если подобная ситуация не возникнет, нет никакой гарантии, что создаваемая *ad hoc* архитектура не будет намного более запутанной и сложной, чем было бы продуманное заранее решение.

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

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

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

CMMI

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

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 следует принять решение, до какого уровня зрелости она намерена дойти, а также необходимо ли проходить официальную сертификацию на соответствие этому уровню.

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

***Структура CMMI* (*ступенчатое представление*)**

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

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

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

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

Иерархия целей представлена на рис. 1.5.

Рис. 1.5. Иерархия целей

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

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

Процессная область

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

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

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

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

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

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

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

– установить бюджет и график проекта;

– идентифицировать риски;

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

– определить необходимые ресурсы;

– установить требования к профессиональным навыкам сотрудников, запланировать тренинги;

– запланировать вовлечение всех заинтересованных лиц;

– создать проектный план.

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

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

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

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

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

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

Практика

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

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

Как уже говорилось, под требования *CMMI* можно подогнать самые разнообразные процессы разработки, однако некоторые сочетания явно не имеют смысла. К примеру, вряд ли можно найти разумное применение процессу *XP*, дополненному формальным документированием требований и сбором метрик.

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

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

• в относительно больших организациях, которые могут позволить себе значительные накладные расходы на внедрение и использование процесса;

• при высокой текучести кадров, когда, тем не менее, необходимо поддерживать скорость разработки и качество продуктов на достойном уровне;

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

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

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

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

Выводы

Мы увидели, что современные процессы, использующиеся в реальных проектах, весьма разнообразны. Каждый из них имеет свои преимущества, которые проявляются в соответствующих условиях. Даже, казалось бы, безнадежно устаревшая водопадная модель совершенна адекватна для некоторых проектов. Каждый процесс обладает

также и рядом характеристик, которые ограничивают область его эффективного использования. Эта ситуация вполне типична для разработки ПО, где уже накоплено множество технологий и методик, но не существует универсального метода, оптимального для любой задачи.

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

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

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

RUP

Один из самых известных процессов, использующих итеративную модель разработки, – *Rational Unified Process* (*RUP*). Он был создан во второй половине 1990-х гг. в компании *Rational Software*. Основными разработчиками были Филипп Крачтен (*Philippe Kruchten*), Грейди Буч (*Grady Booch*), Джеймс Рамбо (*James Rumbaugh*) и Айвар Якобсон (*Ivar Jacobson*). Кстати, последние трое являются также создателями нотации *UML*.

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

Можно выделить следующие основные характеристики процесса *RUP*.

Разработка требований

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

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

Итеративная разработка

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

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

Архитектура

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

итераций необходимо выбирать прецеденты, которые требуют реализации большей части архитектурных компонентов.

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

Жизненный цикл проекта

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

***Начало* (*Inception*)**

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

– определить видение и границы проекта;

– создать экономическое обоснование (*business case*);

– идентифицировать большую часть прецедентов использования и подробно описать несколько ключевых прецедентов;

– найти хотя бы одно возможное архитектурное решение;

– оценить бюджет, график и риски проекта.

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

***Проектирование* (*Elaboration*)**

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

– детальное описание большей части прецедентов использования;

– создание оттестированной (при помощи архитектурно значимых прецедентов использования) базовой архитектуры;

– снижение основных рисков и уточнение бюджета и графика проекта.

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

***Построение* (*Construction*)**

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

***Внедрение* (*Transition*)**

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

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

Каждая из стадий создания системы предусматривает выполнение определенного объема работ, которые представляются в виде процессов ЖЦ. Процесс определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

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

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

Среди наиболее известных стандартов можно выделить следующие:

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

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

• *Custom Development Method* (методика *Oracle*) по разработке прикладных информационных систем – технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением *Oracle*. Применяется *CDM* для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий «быстрой разработки» (*Fast Track*) или «облегченного подхода», рекомендуемых в случае малых проектов.

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

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

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

1.2. Интегрированная среда разработки программного обеспечения

Интегрированная среда разработки, ИСР (англ. *IDE*, *Integrated Development Environment* или *Integrated Debugging Environment*) – система программных средств, используемая программистами для разработки программного обеспечения.

Обычно среда разработки включает:

– текстовый редактор;

– компилятор и (или) интерпретатор;

– средства автоматизации сборки;

– отладчик.

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

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

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

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

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

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

– добавление новых функций;

– известные проблемы, которые нужно устранить;

– предполагаемые к реализации классы;

– места размещения кода обработчиков ошибок;

– напоминания о необходимости переработки участка кода.

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

модействие с другими инструментальными средствами (программным отладчиком-симулятором, внутрисхемным эмулятором, эмулятором постоянного запоминающего устройства и программатором).

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

При традиционном подходе начальный этап написания программы строится следующим образом:

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

• По завершении набора работа с текстовым редактором прекращается и запускается кросс-компилятор.

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

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

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

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

Работа в интегрированной среде дает программисту:

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

– иметь автоматическую диагностику выявленных при компиляции ошибок, когда исходный текст программы, доступный редактированию, выводится одновременно с диагностикой в многооконном режиме;

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

– минимум перекомпиляции. Ей подвергаются только редактировавшиеся модули;

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

– возможность подключения к оболочке практически любых программных средств.

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

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

1.3. Системы управления версиями

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

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

Локальные системы контроля версий

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

Для того чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий (рис. 1.6).

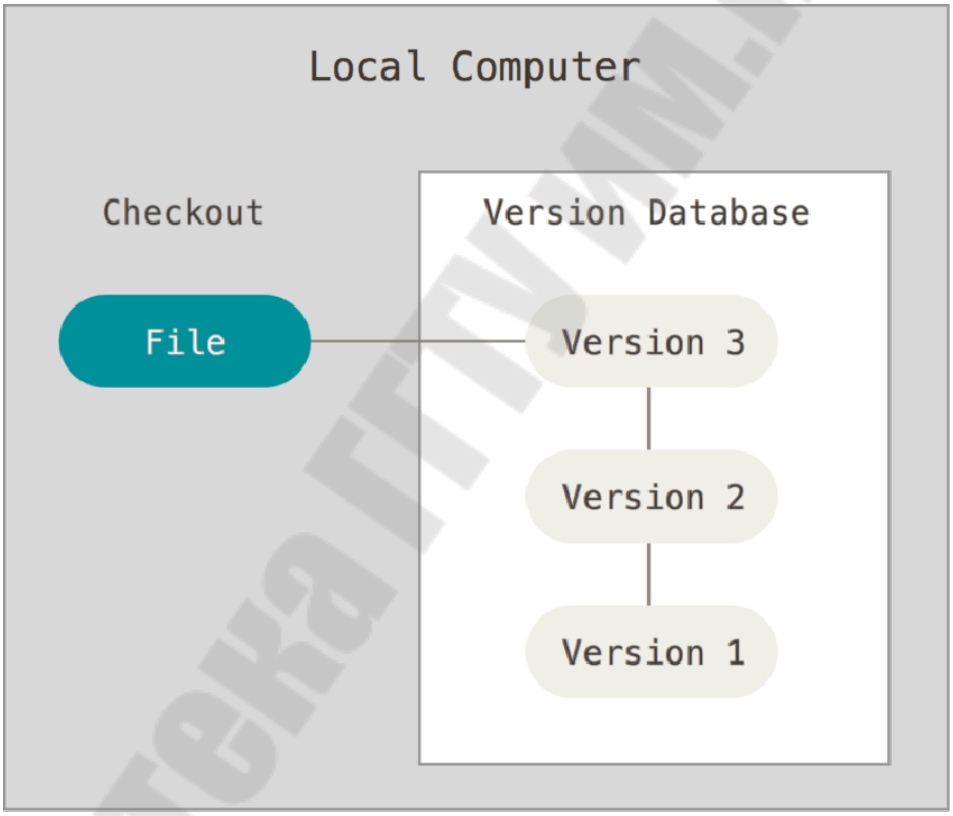

Рис. 1.6. Локальный контроль версий

Одной из популярных СКВ была система *RCS*, которая и сегодня распространяется со многими компьютерами. *RCS* хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые, она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьезная проблема, с которой сталкиваются люди, – это необходимость взаимодействовать с другими разработчиками. Для того чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как *CVS*, *Subversion* и *Perforce*, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет (рис. 1.7).

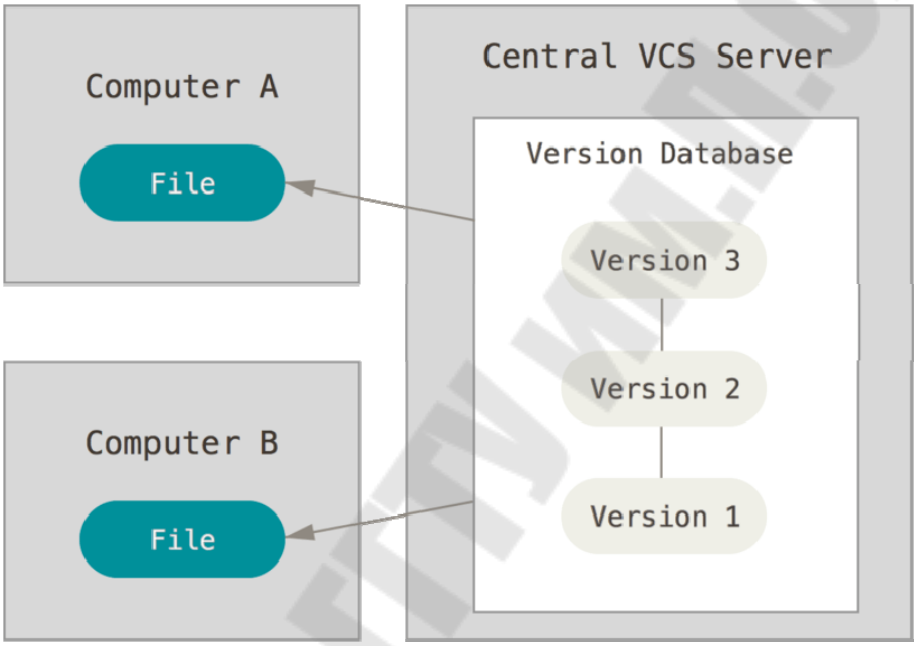

Рис. 1.7. Централизованный контроль версий

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

Несмотря на это, данный подход имеет и серьезные минусы. Самый очевидный минут – это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жесткий диск, на котором хранится центральная база данных, поврежден, а своевременные бэкапы отсутствуют, вы поте-

ряете все всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять все.

Распределенные системы контроля версий

Здесь в игру вступают распределенные системы контроля версий (РСКВ) (рис. 1.8). В РСКВ (таких как *Git*, *Mercurial*, *Bazaar* или *Darcs*) клиенты не просто скачивают снимок всех файлов (состояние файлов на определенный момент времени) – они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, «умрет», любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

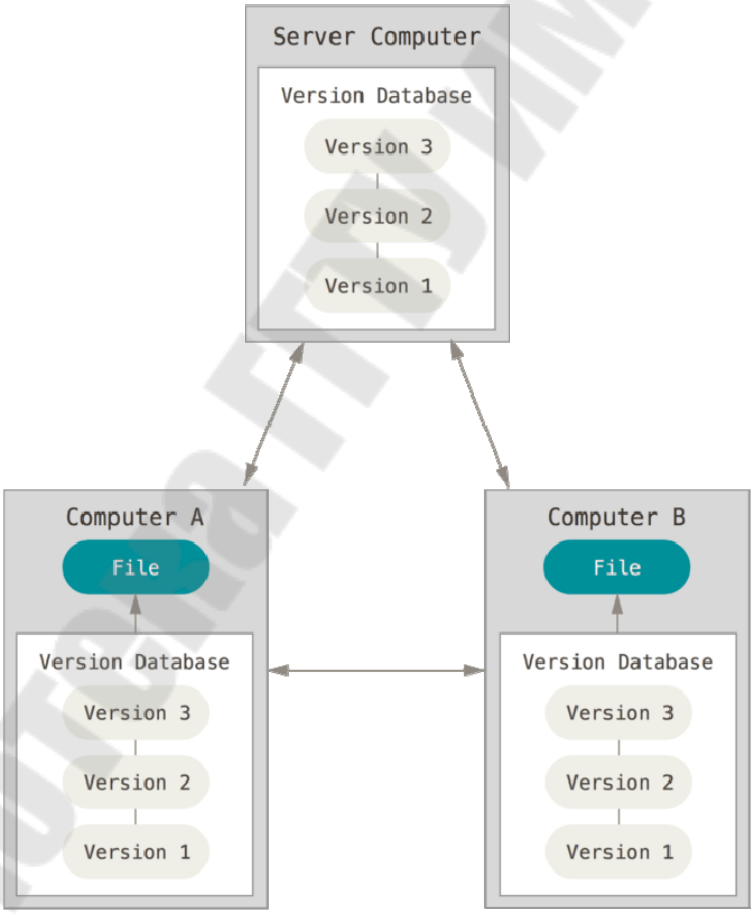

Рис. 1.8. Распределенный контроль версий

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удаленными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы

единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

1.4. Работа с репозиториями

С момента своего появления в 2005 г., *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: 1) вы можете взять локальный каталог, который в настоящее время не находится под версионным контролем, и превратить его в репозиторий *Git*; 2) 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 создает в текущем каталоге новый подкаталог с именем *.git*, содержащий все необходимые файлы репозитория, – структуру *Git*-репозитория. На этом этапе ваш проект еще не находится под версионным контролем. Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений.

Добиться этого, вы сможете, запустив команду *git add* несколько раз, указав индексируемые файлы, а затем выполнив *git commit*:

$ *git add *.c*
$ *git add LICENSE*
$ *git commit -m 'Initial project version'*

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

Клонирование существующего репозитория

Для получения копии существующего *Git*-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду *git clone*. Если вы знакомы с другими системами контроля версий, такими как *Subversion*, то заметите, что команда называется *clone*, а не *checkout*. Это важное различие – вместо того чтобы просто получить рабочую копию, *Git* получает копию практически всех данных, которые есть на сервере. При выполнении *git clone* с сервера забирается (*pulled*) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, 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, помещенные под версионный контроль, будут сохранены).

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

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

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

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

Эта команда делает все 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 неизмененными, измененными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы – это те файлы, о которых знает *Git*.

Неотслеживаемые файлы – это все 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 неизмененными, потому что *Git* только что их извлек, и вы ничего пока не редактировали.

Как только вы отредактируете файлы, *Git* будет рассматривать их как измененные, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется (рис. 1.9).

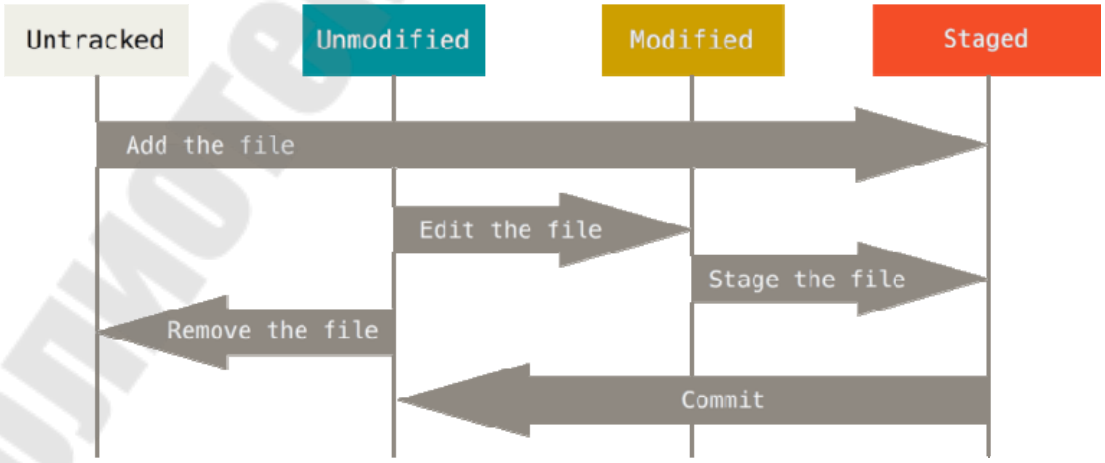

Рис. 1.9. Жизненный цикл состояний файлов

Определение состояния файлов

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

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас – чистый рабочий каталог, другими словами – в нем нет отслеживаемых измененных файлов. *Git* также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам, на какой ветке вы находитесь, и 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*, то увидите свой неотслеживаемый файл вот таким:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README
nothing added to commit but untracked files present (use "git add" to track)
```

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

Отслеживание новых файлов

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

$ *git add README*

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

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

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

Индексация измененных 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 изменен в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду *git add*. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например, для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект». Выполним *git add*, чтобы проиндексировать *CONTRIBUTING.md*, а затем снова выполним *git status*:

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

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в *CONTRIBUTING.md* до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка еще 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 непроиндексированный 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* попадет 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 придется 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*, то получите гораздо более упрощенный вывод:

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

Новые неотслеживаемые файлы помечены ?? слева от них, файлы добавленные в отслеживаемые помечены *A*, отредактированные файлы помечены *M* и т. д. 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 еще 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 серьезно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

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

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

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

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

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

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

Glob-шаблоны представляют собой упрощенные 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, разделенные дефисом ([0-9]), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звездочки, чтобы указать на вложенные каталоги: *a*/**/*z* соответствует *a*/*z*, *a*/*b*/*z*, *a*/*b*/*c*/*z* и т. д.

Вот еще один пример файла *.gitignore*:

Исключить все файлы с расширением *.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 то, 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 еще 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 удаленные строки – патч как он есть.

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

$ *git status*

On branch master

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

Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
modified: *README*
Changes not staged for commit:
(*use "git add <file>..." to update what will be committed*)
(*use "git checkout -- <file>..." to discard changes in working directory*)
modified: *CONTRIBUTING.md*

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

$ *git diff*
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8*ebb*991..643*e*24*f* 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 *@@ branch directly, things can get messy.*
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.
If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (*and note in the PR title that it's*

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

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

$ *git diff --staged*
diff --git a/README b/README
new file mode 100644
index 0000000..03902*a*1
--- /dev/null

+++ b/README
@@ -0,0 +1 @@
+My Project

Важно отметить, что *git diff* сама по себе не показывает все изменения, сделанные с последнего коммита, – только те, что еще не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то *git diff* ничего не вернет.

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 непроиндексированных изменений:

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

Также *git diff --cached* – для просмотра проиндексированных изменений (*--staged* и *--cached* – синонимы):

$ *git diff --cached*
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8*ebb*991..643*e*24*f* 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 *@@ branch directly, things can get messy.*
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.
If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's

Коммит изменений

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

$ *git commit*

Эта команда откроет выбранный вами текстовый редактор.

В редакторе будет отображен 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 еще одну пустую строку сверху. Вы можете удалить эти комментарии и набрать свое сообщение или же оставить их для напоминания о том, что вы фиксируете.

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

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

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

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

Запомните, что коммит сохраняет снимок состояния вашего индекса. Все, что вы не проиндексировали, так и висит в рабочем каталоге как измененное; вы можете сделать еще один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже 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* за-

ставляет *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 83e38c7] 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 попадет 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 еще 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 жестком диске, но перестать отслеживать изменения в нем. Это особенно полезно, если вы забыли добавить что-то в файл *.gitignore* и по ошибке проиндексировали, например, большой файл с логами или «кучу» промежуточных файлов компиляции. Чтобы сделать это, используйте опцию *--cached*:

```
$ git rm --cached README
```

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

```
$ git rm log/\*.log
```

Обратите внимание на обратный слеш (\) перед *. Он необходим из-за того, что *Git* использует свой собственный обработчик имен 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 нем 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 трех – это функция для удобства. Важнее другое – вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами *add*/*rm* перед коммитом.

Работа с удаленными репозиториями

Для того чтобы внести вклад в какой-либо *Git*-проект, вам необходимо уметь работать с удаленными репозиториями. Удаленные репозитории представляют собой версии вашего проекта, сохраненные в интернете или еще где-то в сети. У вас может быть несколько удаленных репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удаленными репозиториями, а также отправку и получение данных из них. Управление репозиториями

включает в себя умение как добавлять новые, так и удалять устаревшие репозитории, а также управлять различными удаленными ветками, объявлять их отслеживаемыми или нет и т. д. В данном разделе мы рассмотрим некоторые из этих навыков.

Просмотр удаленных репозиториев

Для того чтобы просмотреть список настроенных удаленных репозиториев, вы можете запустить команду *git remote*. Она выведет названия доступных удаленных репозиториев. Если вы клонировали репозиторий, то увидите, как минимум, *origin* – имя по умолчанию, которое *Git* дает 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 удаленного репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удаленными репозиториями в случае совместной работы нескольких пользователей вывод команды может выглядеть примерно так:

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
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 дает никакой информации о правах доступа.

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

Добавление удаленных репозиториев

В предыдущих параграфах мы уже упоминали и приводили примеры добавления удаленных репозиториев, сейчас рассмотрим эту операцию подробнее. Для того чтобы добавить удаленный 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 нет у вас, вы можете выполнить команду *git fetch pb*:

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

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

Получение изменений из удаленного репозитория – Fetch и Pull

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

$ *git fetch [remote-name]*

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

Когда вы клонируете репозиторий, команда *clone* автоматически добавляет этот удаленный 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 удаленной ветки, то вы можете использовать команду *git pull*, чтобы автоматически получить изменения из удаленной 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 удаленной 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 удаленный репозиторий. Команда для этого действия простая: *git push <remote-name> <branch-name>*. Чтобы отправить вашу ветку *master* на сервер *origin* (повторимся, что клонирование

обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

$ *git push origin master*

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду *push*. Если вы и кто-то еще одновременно клонируете, и затем он выполняет команду *push*, а после него выполнить команду *push* попытаетесь вы, то ваш *push* точно будет отклонен. Вам придется 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 удаленных 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 выдает *URL* удаленного 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 удаленного сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдает 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, то можете увидеть гораздо большее количество информации от *git remote show*:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
   master                    tracked
   dev-branch                tracked
   markdown-strip            tracked
   issue-43                  new (next fetch will store in remotes/origin)
   issue-45                  new (next fetch will store in remotes/origin)
   refs/remotes/origin/issue-11  stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
  dev-branch merges with remote dev-branch
  master     merges with remote master
  Local refs configured for 'git push':
  dev-branch          pushes to dev-branch          (up to date)
  markdown-strip      pushes to markdown-strip      (up to date)
  master              pushes to master              (up to date)
```

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

Удаление и переименование удаленных репозиториев

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

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также изменит имена удаленных 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 (сменили сервер или больше не используете определенное зеркало, или кто-то перестал вносить изменения), то можете использовать *git remote rm*:

```
$ git remote remove paul
$ git remote
origin
```

При удалении ссылки на удаленный репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, также будут удалены.

SSH-доступ

Если у вас уже есть сервер, к которому все ваши разработчики имеют доступ по *SSH*, проще всего разместить ваш первый репозиторий там, поскольку вам не нужно практически ничего делать (как мы уже обсудили в предыдущем параграфе). 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 учетных записей для членов команды, которым требуются права на запись, то 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 учетные 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 еще нет ключа. По умолчанию пользовательские *SSH*-ключи сохраняются в каталоге *~/.ssh* домашнем каталоге пользователя. Вы можете легко проверить наличие ключа, перейдя в этот каталог и посмотрев его содержимое:

```
$ cd ~/.ssh
$ ls
authorized_keys2  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:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3
schacon@mylaptop.local
```

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

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

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTG
K9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h
9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V
6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDv
jYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIk
jn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

ГЛАВА 2. АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Понятие и виды архитектур программного обеспечения

Построение программной архитектуры осуществляется на основании сопоставления функциональных требований к ИС и (или) их компонентам с функционалом, предоставляемым типовыми классами программных продуктов и (или) программных платформ.

В данном параграфе необходимо указать, какая технология использовалась при построении ИС (например, облачная, клиент-серверная и т. п.).

Также нужно указать тип архитектуры (многоуровневая, мультисервисная и т. п.).

С точки зрения иерархии архитектура классифицируется следующим образом:

– одноуровневая;

– двухуровневая;

– трехуровневая.

С точки зрения распределенности архитектура подразделяется на 3 группы:

– централизованная;

– распределенная;

– гибридная.

С точки зрения наличия внутренних компонентов архитектура составляет 2 группы:

– монолитная;

– компонентная.

Для многоуровневой архитектуры архитектурная модель должна включать слои и их описание (при наличии данного слоя):

– слой клиента – интерфейсный программный элемент, предоставляемый конечному пользователю и не имеющий прямых связей с базой данных. Желательно использование веб-браузеров, использование других средств, в том числе «толстых» клиентов, требует согласования;

– слои презентации – формирование клиентского контента, графическое оформление, управление передачей данных;

– слои бизнес-логики – на этих уровнях сосредоточена прикладная часть логики с возможностью модификации со стороны заказчика;

– промежуточный слой – на этих уровнях может выполняться взаимодействие между слоем данных и слоем клиента;

– транзакционный слой – на этих уровнях может выполняться обмен транзакциями между слоями;

– слои обеспечения интеграции – на этих уровнях реализуются интерфейсы взаимодействия с внешними системами и подсистемами;

– слои БПО – на этих уровнях сосредоточена системная часть платформы;

– слои данных – программные элементы, обеспечивающие хранение данных средствами СУБД, ФС, облачных и локальных хранилищ.

Для мультисервисной архитектуры должна быть указана следующая информация:

– список сервисов (желательно в рамках их общепринятой классификации *ITIL*)

– описание сервисов;

– описание средств обмена между сервисами;

– описание принципов обеспечения отказоустойчивости при недоступности сервисов;

– принципы построения оркестрации и масштабируемости.

Для описания архитектуры рекомендуется использовать стандартные шаблоны (паттерны).

2.2. Проектирование и моделирование системной архитектуры

Системная архитектура отражает фундаментальную организацию информационной системы, реализованную в ее компонентах, связях этих компонентов друг с другом и внешней средой и принципах, определяющих структуру и развитие системы.

Системную архитектуру ИС разрабатывают под систему (подсистему, компонент, модуль).

Решения, отраженные в архитектуре ИС, должны обеспечивать построение ИС, соответствующей следующим принципам:

– открытость – возможность системы допускать замену любого элемента системы без пересмотра системной архитектуры;

– модифицируемость – возможность изменения алгоритмов работы системы путем изменения конфигурационных данных;

– масштабируемость – возможность горизонтально и вертикально наращивать ресурсы системы с пропорциональным повышением производительности таким образом, что при этом не возникает необходимости модернизации программного обеспечения системы или проведения структурных изменений системы;

– надежность – возможность системы сохранять во времени в установленных пределах значения всех параметров, характеризующих способность выполнять требуемые функции в заданных режимах и условиях применения;

– тестируемость – возможность установления факта правильного функционирования системы;

– диагностируемость – возможность нахождения неисправной части системы;

– простота обслуживания и эксплуатации – минимальные требования к квалификации и дополнительному обучению эксплуатационного персонала;

– ремонтопригодность – возможность восстановления работоспособности за минимальное время при экономически оправданной стоимости ремонта;

– безопасность – соответствие требованиям промышленной безопасности и технике безопасности;

– защищенность компонентов системы от злоумышленников и неквалифицированных пользователей;

– экономичность – экономическая эффективность в процессе функционирования;

– долговечность – максимальная длительность жизненного цикла системы без существенного морального старения за счет выбора перспективных промышленных стандартов;

– унифицируемость – в состав компонент каждого модуля входят одинаковые элементы программного обеспечения.

Например, хранящиеся и обрабатываемые в ИС данные можно разделить по назначению при выполнении бизнес-процессов и процессов эксплуатации ИС на следующие основные категории:

– нормативно-справочная информация – справочники, классификаторы и реестры;

– метаинформация – данные, описывающие предметную область: состав и характеристики бизнес-процессов, структуру, правила отображение и контроля данных;

– транзакционные данные – данные, в том числе поступающие в систему в виде экземпляров формуляров для отражения в накопительных и аналитических регистрах транзакционных ИС;

– аналитические данные – данные, полученные из ИС и внешних систем и преобразованные для целей их аналитической обработки (многомерные кубы, агрегация), а также результаты аналитической обработки указанных данных;

– статистические данные – данные из экземпляров формуляров после их статистической обработки;

– служебные данные – данные, необходимые для обеспечения функционирования и администрирования ИС, в том числе данные мониторинга, новое и обновленное программное обеспечение ИС.

Структура данных

С точки зрения своей внутренней структуры данные можно разделить на 2 типа:

– структурированные данные – имеющие описания структуры в ИС;

– неструктурированные данные – не имеющие заранее определенной структуры данных, либо неорганизованные в установленном порядке.

Примечание. Если данные даже частично структурированы, в данным случае они все равно будут считаться неструктурированными (несмотря на то, что в некоторых источниках для них может использоваться термин «полуструктурированные»).

Концептуальная модель хранилища данных представляет собой описание главных (основных) сущностей и отношений между ними. Концептуальная модель является отражением предметных областей, в рамках которых планируется построение хранилища данных.

Логическая модель расширяет концептуальную путем определения сущностей для их атрибутов, описаний и ограничений, уточняет состав сущностей и взаимосвязи между ними.

Существует четыре логические модели данных:

– иерархические;

– сетевые;

– реляционные;

– многомерные.

Уровни логической модели

Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

– диаграмма «сущность–связь» (*Entity Relationship Diagram*, *ERD*);

– модель данных, основанная на ключах (*Key Based model*, *KB*);

– полная атрибутивная модель (*Fully Attributed model*, *FA*).

Диаграмма «сущность–связь» представляет собой модель данных верхнего уровня. Она включает сущности и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют основным требованиям, предъявляемым к ИС. Диаграмма «сущность–связь» может включать связи «многие-ко-многим» и не включать описание ключей. Как правило, *ERD* используется для презентаций и обсуждения структуры данных с экспертами предметной области. Для ее описания может быть использована любая из нотаций: *IDEF*1*x*, *UML*, Питера Чена и др.

Модель данных, основанная на ключах, – более подробное представление данных. Она включает описание всех сущностей и первичных ключей и предназначена для представления структуры данных и ключей, которые соответствуют предметной области.

Полная атрибутивная модель – наиболее детальное представление структуры данных: представляет данные в третьей нормальной форме и включает все сущности, атрибуты и связи.

Создание схемы базы данных выполняется на основе конкретной модели данных, например, реляционной модели данных. Для реляционной модели данных даталогическая модель – набор схем отношений, обычно с указанием первичных ключей, а также «связей» между отношениями, представляющих собой внешние ключи.

2.3. Объектно-ориентированный анализ систем

Объектно-ориентированный анализ – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся *OOA*, *OOD* и *OOP*? На результатах *OOA* формируются модели, на которых основывается *OOD*; *OOD*, в свою очередь, создает фундамент для окончательной реализации системы с использованием методологии *OOP*.

Парадигмы программирования

Дженкинс и Глазго считают, что «в большинстве своем программисты используют в работе один язык программирования и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче» [2]. Бобров и Стефик так определили понятие стиля программирования: «Это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле» [3]. Эти же авторы выявили пять основных разновидностей стилей программирования.

Невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила, а для вычислительных задач – процедурно-ориентированный. По нашему опыту, объектно-ориентированный стиль является наибо-

лее приемлемым для широчайшего круга приложений; действительно, эта парадигма часто служит архитектурным фундаментом, на котором мы основываем другие парадигмы.

Каждый стиль программирования имеет свою концептуальную базу. Каждый стиль требует своего умонастроения и способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база – это *объектная модель.* Она имеет четыре главных элемента:

– абстрагирование;

– инкапсуляция;

– модульность;

– иерархия.

Эти элементы являются *главными* в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных имеются еще три дополнительных элемента:

– типизация;

– параллелизм;

– сохраняемость.

Называя их *дополнительными*, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

Без такой концептуальной основы вы можете программировать на языке типа *Smalltalk*, *Object Pascal*, *C++*, *CLOS*, *Eiffel* или *Ada*, но из-под внешней красоты будет выглядывать стиль *FORTRAN*, *Pascal* или *C*. Выразительная способность объектно-ориентированного языка будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов справиться со сложностью решаемых задач.

Абстрагирование

Смысл абстрагирования. Абстрагирование является одним из основных методов, используемых для решения сложных задач. Хоар считает, что «абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий» [4]. Шоу определила это понятие так: «Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны» [4]. Берзинс, Грей и Науман рекомендовали, чтобы «идея квалифицировалась как абстракция только,

если она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации» [5]. Суммируя эти разные точки зрения, получим нижеприведенное определение абстракции.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Абельсон и Суссман назвали такое разделение смысла и реализации *барьером абстракции* [5], который основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше [6]. Мы считаем полезным еще один дополнительный принцип, называемый *принципом наименьшего удивления,* согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

По мнению Сейдвица и Старка, «существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование» [7]. Мы стараемся строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.

Клиентом называется любой объект, использующий ресурсы другого объекта (называемого *сервером*). Мы будем характеризовать поведение объекта услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее, которую Мейер назвал *контрактной моделью* программирования [8]: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом.

Другими словами, этот контракт определяет *ответственность* объекта – то поведение, за которое он отвечает [9].

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения. Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется *протоколом.* Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.

Центральной идеей абстракции является понятие инварианта. *Инвариант* – это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать *предусловия* (инварианты предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства, и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация. Как мы увидим далее, некоторые языки имеют средства для работы с исключительными ситуациями: объекты могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты, которые, в свою очередь, могут принять на себя перехват исключения и справиться с проблемой.

Заметим, что понятия *операция*, *метод* и *функция-член* происходят от различных традиций программирования (*Ada*, *Smalltalk* и *C++* соответственно). Фактически они обозначают одно и то же и в дальнейшем будут взаимозаменяемы.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом

подпрограмм и с выполнением операторов. Стиль программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые, в свою очередь, вызывают другие правила, и т. д. Объектно-ориентированный стиль программирования связан с воздействием на объекты (в терминах *Smalltalk* с *передачей объектам сообщений*). Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

Примеры абстракций. Для иллюстрации вышеуказанного приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на выделении абстракций для конкретной задачи, сколько на способе выражения абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы парниковой установки – очень ответственное дело, зависящее как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность (показатель pH) и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в такой задаче – датчик. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено, так что мы должны иметь датчики температуры воды и воздуха, влажности, pH, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры – это объект, который способен измерять температуру там, где он расположен. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число градусов по Фаренгейту, Цельсию или Кельвину. Что такое местоположение датчика? Это некоторое идентифицируемое место в теплице, температуру в котором нам необходимо знать; таких мест, вероятно, немного. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и это отличает его от других датчиков.

Теперь можно задать вопрос о том, каковы обязанности датчика температуры? Мы решаем, что датчик должен знать температуру в своем местонахождении и сообщать ее по запросу. Какие же действия может выполнять по отношению к датчику клиент? Мы принимаем решение о том, что клиент может калибровать датчик и получать от него значение текущей температуры.

Для демонстрации проектных решений будет использован язык *C++*. Итак, вот описания, задающие абстрактный датчик температуры на *C++*:

```
// Температура по Фаренгейту
typedef float Temperature;
// Число, однозначно определяющее положение датчика
typedef unsigned int Location;
class TemperatureSensor {
public:
TemperatureSensor (Location);
~TemperatureSensor();
void calibrate(Temperature actualTemperature);
Temperature currentTemperature() const;
private:...};
```

Здесь два оператора определения типов ***Temperature*** и ***Location*** вводят удобные псевдонимы для простейших типов, и это позволяет нам выражать свои абстракции на языке предметной области. К сожалению, конструкция ***typedef*** не определяет нового типа данных и не обеспечивает его защиты. Например, следующее описание в *C++*: «***typedef int Count;***» просто вводит синоним для примитивного типа *int*. Как мы увидим в следующем разделе, другие языки, такие как *Ada* и *Eiffel*, имеют более изощренную семантику в отношении строгой типизации базовых типов. ***Temperature*** – это числовой тип данных в формате с плавающей точкой для записи температур в шкале Фаренгейта. Значения типа *Location* обозначают места фермы, где могут располагаться температурные датчики.

Класс ***TemperatureSensor*** – это только спецификация датчика; настоящая его начинка скрыта в его закрытой (*private*) части. Класс ***TemperatureSensor*** – это еще не объект. Собственно, датчики – это его *экземпляры,* и их нужно создать, прежде чем с ними можно будет оперировать. Например, можно написать так:

Temperature temperature;
TemperatureSensor greenhouse1Sensor(1);
TemperatureSensor greenhouse2Sensor(2);
temperature = greenhouse1Sensor.currentTemperature();

Рассмотрим инварианты, связанные с операцией *currentTemperature*. Предусловие включает предположение, что датчик установлен в правильным месте в теплице, а постусловие – что датчик возвращает значение температуры в градусах Фаренгейта.

До сих пор мы считали датчик пассивным: кто-то должен запросить у него температуру, и тогда он ответит. Однако есть и другой, столь же правомочный подход. Датчик мог бы активно следить за температурой и извещать другие объекты, когда ее отклонение от заданного значения превышает заданный уровень. Абстракция от этого меняется мало: всего лишь несколько иначе формулируется ответственность объекта. Какие новые операции нужны ему в связи с этим? Обычной идиомой для таких случаев является обратный вызов. Клиент предоставляет серверу функцию (функцию обратного вызова), а сервер вызывает ее, когда считает нужным. Здесь нужно написать что-нибудь вроде:

class ActiveTemperatureSensor {public:
ActiveTemperatureSensor (Location,
*void (*f)(Location, Temperature));*
~ActiveTemperatureSensor();
void calibrate(Temperature actualTemperature);
void establishSetpoint(Temperature setpoint,
Temperature delta);
Temperature currentTemperature() const;
private:...};

Новый класс ***ActiveTemperatureSensor*** стал лишь чуть сложнее, но вполне адекватно выражает новую абстракцию. Создавая экземпляр датчика, мы передаем ему при инициализации не только место, но и указатель на функцию обратного вызова, параметры которой определяют место установки и температуру. Новая функция установки ***establishSetpoint*** позволяет клиенту изменять порог срабатывания датчика температуры, а ответственность датчика состоит в том, чтобы вызывать функцию обратного вызова каждый раз, когда текущая температура ***actualTemperature*** отклоняется от ***setpoint*** больше, чем на ***delta***. При этом клиенту становится известно место срабатывания и температура в нем, а дальше уже он сам должен знать, что с этим делать.

Заметьте, что клиент по-прежнему может запрашивать температуру по собственной инициативе. Но что если клиент не произведет инициализацию, например, не задаст допустимую температуру? При проектировании мы обязательно должны решить этот вопрос, приняв какое-нибудь разумное допущение: пусть считается, что интервал допустимых изменений температуры бесконечно широк.

Как именно класс ***ActiveTemperatureSensor*** выполняет свои обязательства, зависит от его внутреннего представления и не должно интересовать внешних клиентов. Это определяется реализацией его закрытой части и функций-членов.

Рассмотрим теперь другой пример абстракции. Для каждой выращиваемой культуры должен быть задан план выращивания, описывающий изменение во времени температуры, освещения, подкормки и ряда других факторов, обеспечивающих высокий урожай. Поскольку такой план является частью предметной области, вполне оправдана его реализация в виде абстракции.

Для каждой выращиваемой культуры существует свой отдельный план, но общая форма планов у всех культур одинакова. Основу плана выращивания составляет таблица, сопоставляющая моментам времени перечень необходимых действий. Например, для некоторой культуры на 15-е сутки роста план предусматривает поддержание в течение 16 ч температуры 78 F, из них 14 ч – с освещением, а затем понижение температуры до 65 F на остальное время суток. Кроме того, может потребоваться внесение удобрений в середине дня, чтобы поддержать заданное значение кислотности.

Таким образом, план выращивания отвечает за координацию во времени всех действий, необходимых при выращивании культуры. Наше решение заключается в том, чтобы не поручать абстракции плана само выполнение плана, – это будет обязанностью другой абстракции. Так мы ясно разделим понятия между различными частями системы и ограничим концептуальный размер каждой отдельной абстракции.

С точки зрения интерфейса объекта-плана клиент должен иметь возможность устанавливать детали плана, изменять план и запрашивать его. Например, объект может быть реализован с интерфейсом «человек-компьютер» и ручным изменением плана. Объект, который содержит детали плана выращивания, должен уметь изменять сам себя. Кроме того, должен существовать объект-исполнитель плана, умеющий читать план. Как видно из дальнейшего описания, ни один

объект не обособлен, а все они взаимодействуют для обеспечения общей цели. Исходя из такого подхода, определяются границы каждого объекта-абстракции и протоколы их связи.

На *C++* план выращивания будет выглядеть следующим образом. Сначала введем новые типы данных, приближая наши абстракции к словарю предметной области (день, час, освещение, кислотность, концентрация):

```
// Число, обозначающее день года
typedef unsigned int Day;
// Число, обозначающее час дня
typedef unsigned int Hour;
// Булевский тип
enum Lights {OFF, ON};
// Число, обозначающее показатель кислотности в диапазоне от 1 до 14
typedef float pH;
// Число, обозначающее концентрацию в процентах: от 0 до 100
typedef float Concentration;
```

Далее, в тактических целях, опишем следующую структуру:

```
// Структура, определяющая условия в теплице
struct Condition {
Temperature temperature;
Lights lighting;
pH acidity;
Concentration concentration;
};
```

Мы использовали структуру, а не класс, поскольку *Condition* – это просто механическое объединение параметров, без какого-либо внутреннего поведения, и более богатая семантика класса здесь не нужна.

Наконец, вот и план выращивания:

```
class GrowingPlan (
public:
GrowingPlan (char *name);
virtual ~GrowingPlan();
void clear();
virtual void establish(Day, Hour, const Condition&);
const char* name() const;
```

const Condition& *desiredConditions*(*Day*, *Hour*) *const*;
protected:...};

Заметьте, что мы предусмотрели одну новую обязанность: каждый план имеет имя, и его можно устанавливать и запрашивать. Кроме того, заметьте, что операция ***establish*** описана как *virtual* для того, чтобы подклассы могли ее переопределять.

В открытую (*public*) часть описания вынесены конструктор и деструктор объекта (определяющие процедуры его порождения и уничтожения), две процедуры модификации (очистка всего плана *clear* и определение элементов плана *establish*) и два селектора-определителя состояния (функции *name* и ***desiredCondition***). Мы опустили в описании закрытую часть класса, заменив ее многоточием, поскольку сейчас нам важны внешние ответственности, а не внутреннее представление класса.

Инкапсуляция

Хотя мы описывали нашу абстракцию ***GrowingPlan*** как сопоставление действий моментам времени, она не обязательно должна быть реализована буквально как таблица данных. Действительно, клиенту нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока тот соблюдает свои обязательства. На самом деле абстракция объекта всегда предшествует его реализации. А после того, как решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов. Как мудро замечает Ингалс: «Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части» [5]. В то время как абстракция «помогает людям думать о том, что они делают», инкапсуляция «позволяет легко перестраивать программы» [5].

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации, т. е. маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как функции корней растения или химию клеточных стенок. Аналогичным образом при проектировании базы данных принято писать программы так, чтобы они не зависели от физического представления данных; вместо этого

сосредотачиваются на схеме, отражающей логическое строение данных [6]. В обоих случаях объекты защищены от деталей реализации объектов более низкого уровня.

Дисков прямо утверждает, что «абстракция будет работать только вместе с инкапсуляцией» [7]. Практически это означает наличие двух частей в классе: интерфейса и реализации. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бритон и Парнас назвали такие детали «тайнами абстракции» [7]. Итак, изложим определение инкапсуляции.

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Примеры инкапсуляции. Вернемся к примеру гидропонного тепличного хозяйства. Еще одной из ключевых абстракций данной предметной области является нагреватель, поддерживающий заданную температуру в помещении. Нагреватель является абстракцией низкого уровня, поэтому можно ограничиться всего тремя действиями с этим объектом: включение, выключение и запрос состояния. Нагреватель не должен отвечать за поддержание температуры, это будет поведением более высокого уровня, совместно реализуемым нагревателем, датчиком температуры и еще одним объектом. Мы говорим о поведении *более высокого уровня,* потому что оно основывается на простом поведении нагревателя и датчика, добавляя к ним кое-что еще, а именно – *гистерезис* (или запаздывание), благодаря которому можно обойтись без частых включений и выключений нагревателя в состояниях, близких к граничным. Приняв такое решение о разделении ответственности, мы делаем каждую абстракцию более цельной.

Как всегда, начнем с типов:

// Булевский тип*enum Boolean* {*FALSE*, *TRUE*};

В дополнение к трем предложенным выше операциям нужны обычные метаоперации создания и уничтожения объекта (конструктор и деструктор). Поскольку в системе может быть несколько нагревателей, мы будем при создании каждого из них сообщать ему место, где он установлен, как мы делали это с классом датчиков температуры ***TemperatureSensor***. Итак, вот класс ***Heater*** для абстрактных нагревателей, написанный на *C++*:

```
class Heater {
public:
Heater(Location);
~Heater();
void turnOn();
void tum0ff();
Boolean is0n() const;
private:};
```

Вот и все, что посторонним надо знать о классе ***Heater***. Внутренность класса – это совсем другое дело. Предположим, проектировщики аппаратуры решили разместить управляющие компьютеры вне теплицы (где слишком жарко и влажно), и соединить их с датчиками и исполнительными устройствами с помощью последовательных интерфейсов. Разумно ожидать, что нагреватели будут коммутироваться с помощью блока реле, а оно будет управляться командами, поступающими через последовательный интерфейс. Скажем, для включения нагревателя передается текстовое имя команды, номер места нагревателя и еще одно число, используемое как сигнал включения нагревателя.

Вот класс, выражающий абстрактный последовательный порт:

```
class SerialPort {
public:
SerialPort();
~SerialPort();
void write(char*);
void write(int);
static SerialPort ports[10];
private:};
```

Экземпляры этого класса будут настоящими последовательными портами, в которые можно выводить строки и числа.

Добавим еще три параметра в класс ***Heater***:

```
class Heater {
public:...
protected:
const Location repLocation;
Boolean repIsOn;
SerialPort* repPort;
};
```

Эти параметры ***repLocation***, ***repIsOn***, ***repPort*** образуют его инкапсулированное состояние. Правила *C++* таковы, что при компиляции программы, если клиент попытается обратиться к этим параметрам напрямую, будет выдано сообщение об ошибке.

Определим теперь реализации всех операций этого класса:

```
Heater::Heater(Location 1)
: repLocation(1),repIsOn(FALSE),repPort(&SerialPort::ports[l]) {}
Heater::Heater() {}
void Heater::turnOn(){
if (!repls0n) {
repPort->write("*");
repPort->write(repLocation);
repPort->write(1);
repIsOn = TRUE;
}
}
void Heater::turn0ff(){
if (repIsOn) {
repPort->write("*");
repPort->write(repLocation);
repPort->write(0);
repIsOn = FALSE;
}
}
Boolean Heater::is0n() const{
return repIsOn;
}
```

Такой стиль реализации типичен для хорошо структурированных объектно-ориентированных систем: классы записываются экономно, поскольку их специализация осуществляется через подклассы.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через фиксированную область памяти. Нет необходимости изменять интерфейсную часть класса – достаточно переписать реализацию. Согласно правилам *C++*, после этого придется перекомпилировать измененный класс, но не другие объекты, если только они не зависят от временных и пространственных характеристик прежнего кода (что крайне нежелательно и совершенно не нужно).

Обратимся теперь к реализации класса ***GrowingPlan***. Как было сказано, это в сущности временной график действий. Вероятно, лучшей реализацией его был бы словарь пар «время-действие» с открытой хеш-таблицей. Нет смысла запоминать действия час за часом, они происходят не так часто, а в промежутках между ними система может интерполировать ход процесса.

Инкапсуляция скроет от посторонних взглядов два секрета: то, что в действительности график использует открытую хеш-таблицу, и то, что промежуточные значения интерполируются. Клиенты вольны думать, что они получают данные из почасового массива значений параметров.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации этих условий для конкретных языков программирования является предметом постоянных обсуждений. Так, *Smalltalk* обеспечивает защиту от прямого доступа к экземплярам другого класса, обнаруживая такие попытки во время компиляции. В то же время *Object Pascal* не инкапсулирует представление класса, так что ничто в этом языке не предохраняет клиента от прямых ссылок на внутренние поля другого объекта. Язык *CLOS* занимает в этом вопросе про-

межуточную позицию, возлагая все обязанности по ограничению доступа на программиста. В этом языке все *слоты* могут сопровождаться атрибутами ***:reader***, ***:writer*** и ***:accessor***, разрешающими соответственно чтение, запись или полный доступ к данным (т. е. и чтение, и запись). При отсутствии атрибутов слот полностью инкапсулирован. По соглашению признание того, что некоторая величина хранится в слоте, рассматривается как нарушение абстракции, так что хороший стиль программирования на *CLOS* требует, чтобы при публикации интерфейса класса, документировались бы только имена его функций, а тот факт, что слот имеет функции полного доступа, должен скрываться [7]. В языке *C++* управление доступом и видимостью более гибко. Члены класса могут быть отнесены к открытой, закрытой или защищенной частям. Открытая часть доступна для всех объектов; закрытая часть полностью закрыта для других объектов; защищенная часть видна только экземплярам данного класса и его подклассов. Кроме того, в *C++* существует понятие «друзей» (*friends*), для которых открыта закрытая часть.

Скрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно; правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться. Инкапсуляция не спасает от глупости; она, как отметил Страуструп, «защищает от ошибок, но не от жульничества» [6]. Разумеется, язык программирования тут вообще ни при чем; разве что операционная система может ограничить доступ к файлам, в которых описаны реализации классов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации.

Модульность

Понятие модульности. По мнению Майерса, «разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом». В некоторых языках программирования, например, в *Smalltalk*, модулей нет, и классы составляют единственную физическую основу декомпозиции. В других языках, включая *Object Pascal*, *C++*, *Ada*, *CLOS*, модуль – это самостоятельная

языковая конструкция. В этих языках классы и объекты составляют логическую структуру системы, они помещаются в модули, образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Согласно Барбаре Лисков, «модульность – это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями». Мы будем пользоваться определением Парнаса: «Связи между модулями – это их представления друг о друге» [8]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, модульность и инкапсуляция «ходят рука об руку». В разных языках программирования модульность поддерживается по-разному. Например, в *C++* модулями являются раздельно компилируемые файлы. Для *C*/*C++* традиционно помещение интерфейсной части модулей в отдельные файлы с расширением *.h* (так называемые *файлы-заголовки).* Реализация, т. е. текст модуля, хранится в файлах с расширением .c (в программах на *C++* часто используются расширения .ee, .cp и .cpp). Связь между файлами объявляется директивой макропроцессора *#include*. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка. В языке *Object Pascal* принцип модульности формализован несколько строже. В этом языке определен особый синтаксис для интерфейсной части и реализации модуля (*unit*). Язык *Ada* идет еще на шаг дальше: модуль (называемый *package*) также имеет две части – спецификацию и тело. Но в отличие от *Object Pascal* допускается раздельное определение связей с модулями для спецификации и тела пакета. Таким образом, допускается, чтобы тело модуля имело связи с модулями, невидимыми для его спецификации.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зельковиц, утверждая: «поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной» [9].

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Такая же ситуация возникает у проектировщиков

бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ (кроме самых тривиальных) лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации своей работы механизм передачи сообщений. В больших системах вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. При этом не только возникает «кошмар» с документированием, но даже просто поиск нужных фрагментов описания становится чрезвычайно труден для пользователя. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что скрытие информации имеет и обратную сторону [6]. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

Модульность позволяет хранить абстракции раздельно.

В традиционном структурном проектировании модульность – это искусство раскладывать подпрограммы «по кучкам» так, чтобы в одну «кучку» попадали подпрограммы, использующие друг друга или изменяемые вместе. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.

На основе имеющегося опыта можно перечислить приемы и правила, которые позволяют составлять модули из классов и объектов наиболее эффективным образом. Бритон и Парнас считают, что «конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны» [6].

Прагматические соображения устанавливают предел этим руководящим указаниям. На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль, и программа перекомпонуется. Перекомпиляция *интерфейсной* части модуля, напротив, более трудоемка. В строго типизированных языках приходится перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и так далее по цепочке. В итоге для очень больших программ могут потребоваться многие часы на перекомпиляцию (если только среда разработки не поддерживает фрагментарную компиляцию), что явно нежелательно. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была наиболее узкой (в пределах обеспечения необходимых связей). Наш стиль программирования требует скрыть все, что только возможно, в реализации модуля. Постепенный перенос описаний из реализации в интерфейсную часть гораздо менее опасен, чем «вычищение» избыточного интерфейсного кода.

Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Парнас, Клеменс и Вейс предложили следующее правило: «Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля» [6]. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности.

Модульность – это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

В процессе разделения системы на модули могут быть полезными два правила. Во-первых, поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использо-

ваться в системе повторно, распределение классов и объектов по модулям должно учитывать это. Во-вторых, многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные – за реализацию. На более крупном уровне такие же соотношения справедливы для отношений между субподрядчиками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между компаниями, участвующими в работе. Изменения в интерфейсе вызывают много «крика и зубовного скрежета», не говоря уже об огромном расходе бумаги, – все эти факторы делают интерфейс крайне консервативным. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу – модуль служит единицей описания и администрирования. Десять модулей вместо одного потребуют в десять раз больше описаний, и поэтому, к сожалению, иногда требования по документированию влияют на декомпозицию проекта (в большинстве случаев негативно). Могут сказываться и требования секретности: часть кода может быть несекретной, а другая – секретной; последняя тогда выполняется в виде отдельного модуля (модулей).

Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры – *независимые* действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули – этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

Примеры модульности. Посмотрим, как реализуется модульность в гидропонной огородной системе. Допустим, вместо закупки специализированного аппаратного обеспечения решено использовать стандартную рабочую станцию с графическим интерфейсом пользователя *GUI* (*Graphical User Interface*). С помощью рабочей станции оператор может формировать новые планы выращивания, модифицировать имеющиеся планы и наблюдать за их исполнением. Так как абстракция плана выращивания – одна из ключевых, создадим модуль, содержащий все, относящееся к плану выращивания. На *C++* нам понадобится примерно такой файл-заголовок (пусть он называется *gplan.h*):

```
// gplan.h
#ifndef _GPLAN_H
#define _GPLAN_H 1
#include "gtypes.h"
#include "except.h"
#include "actions.h"
class GrowingPlan ...
class FruitGrowingPlan ...
class GrainGrowingPlan ...
#endif
```

Здесь мы импортируем в файл три других заголовочных файла с определением интерфейсов, на которые будем ссылаться: *gtypes.h*, *except .h* и *actions.h*. Собственно код классов мы поместим в модуль реализации, в файл с именем *gplan.cpp*.

Мы могли бы также собрать в один модуль все программы, относящиеся к окнам диалога, специфичным для данного приложения. Этот модуль наверняка будет зависеть от классов, объявленных в *gplan.h*, и от других файлов-заголовков с описанием классов *GUI*.

Вероятно, будет много других модулей, импортирующих интерфейсы более низкого уровня. Наконец мы доберемся до главной функции – точки запуска нашей программы операционной системой. При объектно-ориентированном проектировании это, скорее всего, будет самая малозначительная и неинтересная часть системы, в то время как в традиционном структурном подходе головная функция – это краеугольный камень, который держит все сооружение. Мы полагаем, что объектно-ориентированный подход более естественен, поскольку, как замечает Мейер: «…на практике программные системы

предлагают некоторый набор услуг. Сводить их к одной функции можно, но противоестественно. Настоящие системы не имеют верхнего уровня» [6].

Иерархия

Абстракция – вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Ниже дадим определение иерархии.

Иерархия – это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия «*is-a*») и структура объектов (иерархия «*part of*»).

***Примеры иерархии*: *одиночное наследование*.** Важным элементом объектно-ориентированных систем и основным видом иерархии «*is-a*» является упоминавшаяся выше концепция наследования. Наследование означает такое отношение между классами (отношение «родитель/потомок»), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование*). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Часто подкласс достраивает или переписывает компоненты вышестоящего класса.

Семантически наследование описывает отношение типа «*is-a*». Например, медведь есть млекопитающее, дом есть недвижимость и «быстрая сортировка» есть сортирующий алгоритм. Таким образом, наследование порождает иерархию «обобщение – специализация», в которой подкласс представляет собой специализированный частный случай своего суперкласса. «Лакмусовая бумажка» наследования – обратная проверка; так, если *B* не есть *A*, то *B* не стоит производить от *A*.

Рассмотрим теперь различные виды растений, выращиваемых в нашей «огородной системе». Мы уже ввели обобщенное представ-

ление абстрактного плана выращивания растений. Однако разные культуры требуют разных планов. При этом планы для фруктов похожи друг на друга, но отличаются от планов для овощей или цветов. Имеет смысл ввести на новом уровне абстракции обобщенный «фруктовый» план, включающий указания по опылению и сборке урожая. Вот как будет выглядеть на *C++* определение плана для фруктов как наследника общего плана выращивания:

```
// Тип Урожай
typedef unsigned int Yield;
class FruitGrowingPlan : public GrowingPlan {
public:
FruitGrowingPlan(char* name);
virtual ~FruitGrowingPlan();
virtual void establish(Day, Hour, Condition&);
void scheduleHarvest(Day, Hour);
Boolean isHarvested() const;
unsigned daysUntilHarvest() const;
Yield estimatedYield() const;
protected:
Boolean repHarvested;
Yield repYield;
```

Абстракции образуют иерархию.

Это означает, что план выращивания фруктов ***FruitGrowingPlan*** является разновидностью плана выращивания ***GrowingPlan***. В него добавлены параметры ***repHarvested*** и ***repYield***, определены четыре новые функции и переопределена функция *establish*. Теперь мы могли бы продолжить специализацию – например, определить на базе «фруктового» плана «яблочный» класс ***AppleGrowingPlan***.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии «*обобщение – специализация*». Суперклассы при этом отражают наиболее общие, а подклассы – более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. Кокс пишет: «В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться «с нуля». Классы лишаются общности, поскольку каждый программист реализует их по-своему.

Стройность системы достигается тогда только за счет дисциплинированности программистов. Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям – сравнивая новое с чем-то уже известным» [6].

Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в некоем здоровом конфликте. Данфорт и Томлинсон утверждают: «Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов» [6]. Для любого класса обычно существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники. Лисков поэтому отмечает, что существуют три способа нарушения инкапсуляции через наследование: «подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса» [6]. Различные языки программирования по-разному находят компромисс между наследованием и инкапсуляцией; наиболее гибким в этом отношении является *C++*. В нем интерфейс класса может быть разделен на три части: закрытую (*private*), видимую только для самого класса; защищенную (*protected*), видимую также и для подклассов; и открытую (*public*), видимую для всех.

Примеры иерархии*: *множественное наследование*.** В предыдущем примере рассматривалось одиночное наследование, когда подкласс ***FruitGrowingPlan был создан только из одного суперкласса ***GrowingPlan***. В ряде случаев полезно реализовать наследование от нескольких суперклассов. Предположим, что нужно определить класс, представляющий разновидности растений:

```
class Plant {
public:
Plant(char* name, char* species);
virtual ~Plant();
void setDatePlanted(Day);
virtual establishGrowingConditions(const Condition&);
const char* name() const;
const char* species() const;
Day datePlantedt) const;
protected:
char* repName;
```

```
char* repSpecies;
Day repPlanted;
private:...};
```

Каждый экземпляр класса ***plant*** будет содержать имя, вид и дату посадки. Кроме того, для каждого вида растений можно задавать особые оптимальные условия выращивания. Мы хотим, чтобы эта функция переопределялась подклассами, поэтому она объявлена *виртуальной* при реализации в *C++*. Три параметра объявлены как защищенные, т. е. они будут доступны и классу, и подклассам (закрытая часть спецификации доступна только самому классу).

Изучая предметную область, мы приходим к выводу, что различные группы культивируемых растений (цветы, фрукты и овощи) имеют свои особые свойства, существенные для технологии их выращивания. Например, для цветов важно знать времена цветения и созревания семян. Аналогично, время сбора урожая важно для абстракций фруктов и овощей. Создадим два новых класса – цветы **(*Flower*)** и фрукты-овощи **(*FruitVegetable*)**; они оба наследуют от класса ***Plant***. Однако некоторые цветочные растения имеют плоды! Для этой абстракции придется создать третий класс, ***FlowerFruitVegetable***, который будет наследовать от классов ***Flower*** и ***FruitVegetablePlant***.

Чтобы не было избыточности, в данном случае очень пригодится множественное наследование. Сначала давайте опишем отдельно цветы и фрукты-овощи:

```
class FlowerMixin {
public:
FlowerMixin(Day timeToFlower, Day timeToSeed);
virtual ~FlowerMixin();
Day timeToFlower() const;
Day timeToSeed() const;
protected:...
};
class FruitVegetableMixin {
public:
FruitVegetableMixin(Day timeToHarvest);
virtual ~FruitVegetableMixin();
Day timeToHarvest() const;
protected:...
};
```

Мы намеренно описали эти два класса без наследования. Они ни от кого не наследуют и специально предназначены для того, чтобы их *подмешивали* (откуда и имя ***Mixin***) к другим классам. Например, опишем розу:

class Rose : public Plant, public FlowerMixin...

А вот морковь:

class Carrot : public Plant, public FruiteVegetableMixin {};

В обоих случаях классы наследуют от двух суперклассов: экземпляры подкласса ***Rose*** включают структуру и поведение как из класса ***Plant***, так и из класса ***FlowerMixin***. И вот теперь определим вишню, у которой товаром являются как цветы, так и плоды:

class Cherry : public Plant, public FlowerMixin, FruitVegetableMixin...

Множественное наследование – «вещь нехитрая», но оно осложняет реализацию языков программирования. Есть две проблемы – конфликты имен между различными суперклассами и повторное наследование. Первый случай, это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. В *C++* этот вид конфликта должен быть явно разрешен вручную, а в *Smalltalk* берется то, которое встречается первым. Повторное наследование – это когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования и надо решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, в других конфликт решается «волевым порядком», а в *C++* это оставляется на усмотрение программиста. Виртуальные базовые классы используются для запрещения дублирования повторяющихся структур, в противном случае в подклассе появятся копии полей и функций и потребуется явное указание происхождения каждой из копий.

Множественным наследованием часто злоупотребляют. Например, сладкая вата – это частный случай сладости, но никак не ваты. Применяйте ту же «лакмусовую бумажку»: если *B* не есть *A*, то ему не стоит наследовать от *A*. Часто плохо сформированные структуры множественного наследования могут быть сведены к единственному суперклассу плюс агрегация других классов подклассом.

***Примеры иерархии*: *агрегация*.** Если иерархия «*is*-а» определяет отношение «обобщение/специализация», то отношение «*part of*» (часть) вводит иерархию агрегации. Вот пример:

```
class Garden {
public:
Garden();
virtual ~Garden();
protected:
Plant* repPlants[100];
GrowingPlan repPlan;
};
```

Это – абстракция огорода, состоящая из массива растений и плана выращивания.

Имея дело с такими иерархиями, мы часто говорим об уровнях абстракции, которые впервые предложил Дейкстра [7]. В иерархии классов вышестоящая абстракция является обобщением, а нижестоящая – специализацией. Поэтому мы говорим, что класс ***Flower*** находится на более высоком уровне абстракции, чем класс ***Plant***. В иерархии «*part of*» класс находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Так, класс ***Garden*** стоит на более высоком уровне, чем класс ***Plant***.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

В связи с агрегацией возникает проблема владения или принадлежности объектов. В нашем абстрактном огороде одновременно растет много растений, и от удаления или замены одного из них огород не становится другим огородом. Если мы уничтожаем огород, растения остаются (их ведь можно пересадить). Другими словами, огород и растения имеют свои отдельные и независимые сроки жизни; мы достигли этого благодаря тому, что огород содержит не сами объекты ***Plant***, а указатели на них. Напротив, мы решили, что объект ***GrowingPlan*** внутренне связан с объектом ***Garden*** и не существует независимо. План выращивания физически содержится в каждом экземпляре огорода и погибает вместе с ним. Подробнее про семантику владения мы будем говорить в третьей главе.

Типизация

Что такое типизация? Понятие *типа* взято из теории абстрактных типов данных. Дойч определяет тип как «точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов» [8]. Для наших целей достаточно считать, что термины *тип* и *класс* взаимозаменяемы. Тип и класс не вполне одно и то же; в некоторых языках их различают. Например, ранние версии языка *Trellis*/*Owl* разрешали объекту иметь и класс, и тип. Даже в *Smalltalk* объекты классов *SmallInteger*, *LargeNegativeInteger*, *LargePositiveInteger* относятся к одному типу *Integer*, хотя и к разным классам [9]. Большинству «смертных» различать типы и классы просто противно и бесполезно. Достаточно сказать, что класс реализует понятие типа. Тем не менее типы стоит обсудить отдельно, поскольку они выставляют смысл абстрагирования в совершенно другом свете. В частности, отсюда нижеприведенное утверждение.

Типизация – это способ защититься от использования объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием.

Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений. Вегнер замечает, что такой способ контроля существенен для программирования «в большом» [7].

Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения [7]. Деля расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу – есть. Все это примеры сильной типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Примеры сильной и слабой типизации. Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. Например, в *Eiffel* соблюдение правил использования типов контролируется непреклонно – операция не может быть применена к объекту, если она не зарегистрирована в его классе или суперклассе. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в *Smalltalk* типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение

об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения. *C++* тяготеет к сильной типизации, но в этом языке правила типизации можно игнорировать или подавить полностью.

Рассмотрим абстракцию различных типов емкостей, которые могут использоваться в нашей теплице. Вероятно, в ней есть емкости для воды и для минеральных удобрений; хотя первые предназначены для жидкостей, а вторые для сыпучих веществ, они имеют достаточно много общего, чтобы устроить иерархию классов. Начнем с типов:

```
// Число, обозначающее уровень от 0 до 100 %
typedef float Level;
```

Операторы *typedef* в *C++* не вводят новых типов. В частности, и *Level* и *Concentration* – на самом деле другие названия для *float*, и их можно свободно смешивать в вычислениях. В этом смысле *C++* имеет слабую типизацию: значения примитивных типов, таких как *int* или *float*, неразличимы в пределах данного типа. Напротив, *Ada* и *Object Pascal* предоставляют сильную типизацию для примитивных типов. В *Ada* можно объявить самостоятельным типом интервал значений или подмножество с ограниченной точностью.

Строгая типизация предотвращает смешивание абстракций.

Построим теперь иерархию классов для емкостей:

```
class StorageTank {
public:
StorageTank();
virtual ~StorageTank();
virtual void fill();
virtual void startDraining();
virtual void stopDraining();
Boolean isEmpty() const;
Level level() const;
protected:...
};
class WaterTank : public StorageTank{
public:
WaterTank();
virtual ~WaterTank();
virtual void fill();
virtual void startDraining();
```

```
virtual void stopDraining();
void startHeating();
void stopHeating();
Temperature currentTemperature() const;
protected:...
};
class NutrientTank : public StorageTank {
public:
NutrientTank();
virtual ~NutrientTank();
virtual void startDrainingt();
virtual void stopDraining();
protected:...
};
```

Класс ***StorageTank*** – это базовый класс иерархии. Он обеспечивает структуру и поведение общие для всех емкостей: возможность их наполнять или опустошать. Классы ***WaterTank*** (емкость для воды) и ***NutrientTank*** (для удобрений) наследуют свойства ***StorageTank***, частично переопределяют их и добавляют кое-что свое: например, класс ***WaterTank*** вводит новое поведение, связанное с температурой.

Предположим, что мы имеем следующие описания:

```
StorageTank s1, s2;
WaterTank w;
NutrientTank n;
```

Заметьте, переменные, такие, как *s*1, *s*2, *w* или *n* – это не экземпляры соответствующих классов. На самом деле, это просто имена, которыми мы обозначаем объекты соответствующих классов: когда мы говорим «объект *s*1», мы на самом деле имеем ввиду экземпляр ***StorageTank***, обозначаемый переменной *s*1. Мы вернемся к этому тонкому вопросу в третьей главе.

При проверке типов у классов *C++* типизирован гораздо строже. Под этим понимается, что выражения, содержащие вызовы операций, проверяются на согласование типов во время компиляции. Например, следующее правильно:

```
Level l = s1.level();
w.startDrainingt();
n.stopDraining();
```

Действительно, такие селекторы есть в классах, к которым принадлежат соответствующие переменные. Напротив, следующее неправильно и вызовет ошибку компиляции:

```
s1.startHeating(); // Неправильно
n.stopHeating(); // Неправильно
```

Таких функций нет ни в самих классах, ни в их суперклассах. Но следующее:

```
n.fill();
```

совершенно правильно: функции *fill* нет в определении ***NutrientTank***, но она есть в вышестоящем классе.

Итак, сильная типизация заставляет нас соблюдать правила использования абстракций, поэтому она тем полезнее, чем больше проект. Однако у нее есть и теневая сторона. А именно, даже небольшие изменения в интерфейсе класса требуют перекомпиляции всех его подклассов. Кроме того, не имея параметризованных классов, трудно представить себе, как можно было бы создать собрание разнородных объектов. Предположим, что мы хотим ввести абстракцию инвентарного списка, в котором собирается все имущество, связанное с теплицей. Обычная для *C* идиома применима и в *C++*: нужно использовать класс-контейнер, содержащий указатели на *void*, т. е. на объекты произвольного типа.

```
class Inventory {
public:
Inventory();
~Inventory();
void add(void*);
void remove(void*);
void* mostRecent() const;
void apply(Boolean (*)(void*));
private:...
};
```

Операция *apply* – это так называемый итератор, который позволяет применить какую-либо операцию ко всем объектам в списке.

Имея экземпляр класса *Inventory*, мы можем добавлять и уничтожать указатели на объекты любых классов. Но эти действия небезопасны с точки зрения типов – в списке могут оказаться как осязаемые объекты (емкости), так и неосязаемые (температура или план

выращивания), что нарушает нашу абстракцию материального учета. Более того, мы могли бы внести в список объекты классов ***WaterTank*** и ***TemperatureSensor***, и по неосторожности ожидая от функции *mostRecent* объекта класса ***WaterTank***, получили ***StorageTank***.

Вообще говоря, у этой проблемы есть два общих решения. Во-первых, можно сделать контейнерный класс, безопасный с точки зрения типов. Чтобы не манипулировать с нетипизированными указателями *void*, мы могли бы определить инвентаризационный класс, который манипулирует только с объектами класса ***TangibleAsset*** (осязаемого имущества), а этот класс будет подмешиваться ко всем классам, такое имущество представляющим, например, к ***WaterTank***, но не к ***GrowingPlan***. Тем самым можно отсечь проблему первого рода, когда неправомочно смешиваются объекты разных типов. Во-вторых, можно ввести проверку типов в ходе выполнения, для того чтобы знать – с объектом какого типа мы имеем дело в данный момент. Например, в *Smalltalk* можно запрашивать у объектов их класс. В *C++* такая возможность не входила в стандарт до недавнего времени, хотя на практике, конечно, можно ввести в базовый класс операцию, возвращающую код класса (строку или значение перечислимого типа). Однако для этого надо иметь очень серьезные причины, поскольку проверка типа в ходе выполнения ослабляет инкапсуляцию. Как будет показано в следующем параграфе, необходимость проверки типа можно смягчить, используя полиморфные операции.

В языках с сильной типизацией гарантируется, что все выражения будут согласованы по типу. Что это значит, лучше пояснить на примере. Следующие присваивания допустимы:

```
s1 = s2;
s1 = w;
```

Первое присваивание допустимо, поскольку переменные имеют один и тот же класс, а второе – поскольку присваивание идет снизу вверх по типам. Однако во втором случае происходит потеря информации (известная в *C++* как «проблема срезки»), так как класс переменной *w*, ***WaterTank***, семантически богаче, чем класс переменной *s*1, т. е. ***StorageTank***.

Следующие присваивания неправильны:

```
w = s1; // Неправильно
w = n; // Неправильно
```

В первом случае неправильность в том, что присваивание идет сверху вниз по иерархии, а во втором – классы даже не находятся в состоянии подчиненности.

Иногда необходимо преобразовать типы. Например, посмотрите на следующую функцию:

void checkLevel(const StorageTank& s);

Мы можем привести значение вышестоящего класса к подклассу в том и только в том случае, если фактическим параметром при вызове оказался объект класса ***WaterTank***. Или вот еще случай:

if (((WaterTank&)s).currentTemperature() < 32.0)…

Это выражение согласовано по типам, но небезопасно. Если при выполнении программы вдруг окажется, что переменная *s* обозначала объект класса ***NutrientTank***, приведение типа даст непредсказуемый результат во время исполнения. Вообще говоря, преобразований типа надо избегать, поскольку они часто представляют собой нарушение принятой системы абстракций.

Теслер отметил следующие важные преимущества строго типизированных языков:

• Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.

• В большинстве систем процесс «редактирование–компиляция–отладка» утомителен, и раннее обнаружение ошибок просто незаменимо.

• Объявление типов улучшает документирование программ.

• Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы [7].

Языки, в которых типизация отсутствует, обладают большей гибкостью, но даже в таких языках, по мнению Борнинга и Ингалса: «Программисты обычно знают, какие объекты ожидаются в качестве аргументов и какие будут возвращаться» [7]. На практике, особенно при программировании «в большом», надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости по сравнению с нетипизированными языками.

***Примеры типизации*: *статическое и динамическое связывание*.** Сильная и статическая типизация – разные вещи. Строгая типизация следит за соответствием типов, а статическая типизация (иначе называемая *статическим* или *ранним связыванием*) определяет время, когда имена связываются с типами. Статическая связь означает,

что типы всех переменных и выражений известны во время компиляции; *динамическое связывание* (называемое также *поздним связыванием*) означает, что типы неизвестны до момента выполнения программы. Концепции типизации и связывания являются независимыми, поэтому в языке программирования может быть: типизация – сильная, связывание – статическое (*Ada*), типизация – сильная, связывание – динамическое (*C++*, *Object Pascal*), или и типов нет, и связывание динамическое (*Smalltalk*). Язык *CLOS* занимает промежуточное положение между *C++* и *Smalltalk*: определения типов, сделанные программистом, могут быть либо приняты во внимание, либо не приняты.

Прокомментируем это понятие снова примером на *C++*. Вот «свободная», т. е. не входящая в определение какого-либо класса, функция. Свободная функция – функция, не входящая ни в какой класс. В чисто объектно-ориентированных языках, типа *Smalltalk*, свободных процедур не бывает, каждая операция связана с каким-нибудь классом:

void balanceLevels(StorageTank& s1, StorageTank& s2);

Вызов этой функции с экземплярами класса ***StorageTank*** или любых его подклассов в качестве параметров будет согласован по типам, поскольку тип каждого фактического параметра происходит в иерархии наследования от базового класса ***StorageTank***.

При реализации этой функции мы можем иметь что-нибудь вроде:

if (s1.level()> s2.level()) s2.fill();

В чем особенность семантики при использовании селектора *level*? Он определен только в классе ***StorageTank***, поэтому независимо от классов объектов, обозначаемых переменными в момент выполнения, будет использована одна и та же унаследованная ими функция. Вызов этой функции статически связан при компиляции – мы точно знаем, какая операция будет запущена.

Иное дело *fill*. Этот селектор определен в ***StorageTank*** и переопределен в ***WaterTank***, поэтому его придется связывать динамически. Если при выполнении переменная *s*2 будет класса ***WaterTank***, то функция будет взята из этого класса, а если – ***NutrientTank***, то из ***StorageTank***. В *C++* есть специальный синтаксис для явного указания источника; в нашем примере вызов *fill* будет разрешен, соответственно, как ***WaterTank::fill*** или ***StorageTank::fill.*** Так, синтаксис *C++* определяет явную квалификацию имени.

Эта особенность называется *полиморфизмом*: одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять. Противоположность полиморфизму называется *мономорфизмом*; он характерен для языков с сильной типизацией и статическим связыванием (*Ada*).

Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание. Это одно из самых привлекательных свойств объектно-ориентированных языков (после поддержки абстракции), отличающее их от традиционных языков с абстрактными типами данных. И, как мы увидим в следующих главах, полиморфизм играет очень важную роль в объектно-ориентированном проектировании.

Параллелизм

Что такое параллелизм? Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (*поток управления*) – это фундаментальная единица действия в системе. Каждая программа имеет, по крайней мере, один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течение всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

Кроме этого «аппаратного» различия мы будем различать «тяжелую» и «легкую» параллельность по потребности в ресурсах. «Тяжелые» процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. «Легкие» сосуществуют в одном адресном пространстве. «Тяжелые» процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь «легких» процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах. Например, системы *UNIX* предусматривают системный вызов *fork*, который порождает новый

процесс. Системы *Windows NT* и *OS*/2 – многопоточные; кроме того, они обеспечивают программные интерфейсы для создания процессов и манипулирования с ними.

Лим и Джонсон отмечают, что «возможности проектирования параллельности в объектно-ориентированных языках не сильно отличаются от любых других, – на нижних уровнях абстракции параллелизм и *OOP* развиваются совершенно независимо. С *OOP* или без, все традиционные проблемы параллельного программирования сохраняются» [7]. Действительно, создавать большие программы и так непросто, а если они еще и параллельные, то надо думать о возможном простое одного из потоков, неполучении данных, взаимной блокировке и т. д.

К счастью, как отмечают те же авторы далее: «на верхних уровнях *OOP* упрощает параллельное программирование для рядовых разработчиков, пряча его в повторно-используемые абстракции» [7]. Блэк сделал следующий вывод: «объектная модель хороша для распределенных систем, поскольку она неявно разбивает программу на (1) распределенные единицы и (2) сообщающиеся субъекты» [8].

В то время как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов [8]. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется *активным*. Для систем, построенных на основе *OOD*, мир может быть представлен как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров. На этой основе дадим определение параллелизма.

Параллелизм – это свойство, отличающее активные объекты от пассивных.

Примеры параллелизма. Ранее мы обзавелись классом *ActiveTemperatureSensor*, поведение которого предписывает ему периодически измерять температуру и обращаться к известной ему функции вызова, когда температура отклоняется на некоторую величину от установленного значения. Как он будет это делать, мы в тот момент не объяснили. При всех секретах реализации понятно, что это – активный объект и, следовательно, без параллелизма тут не обойтись. В объектно-ориентированном проектировании есть три подхода к параллелизму.

Во-первых, параллелизм – это внутреннее свойство некоторых языков программирования. Так, для языка *Ada* механизм параллельных процессов реализуется как *задача.* В *Smalltalk* есть класс *process*, которому наследуют все активные объекты. Есть много других языков со встроенными механизмами для параллельного выполнения и синхронизации процессов – *Actors*, *Orient* 84/*K*, *ABCL*/1, которые предусматривают сходные механизмы параллелизма и синхронизации. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность «легкого» параллелизма. Например, библиотека *AT&T* для *C++* содержит классы *Shed*, *Timer*, *Task* и т. д. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встраиваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.

Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Для этого надо кое-что знать об аппаратуре. Например, в нашей реализации класса ***ActiveTemperatureSensor*** мы могли бы иметь аппаратный таймер, периодически прерывающий приложение, после чего все датчики измеряли бы температуру и обращались бы, если нужно, к своим функциям вызова.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет «растерзан» на части несколькими независимыми процессами.

Сохраняемость

Любой программный объект существует в памяти и живет во времени. Аткинсон предположил, что есть непрерывное множество продолжительности существования объектов: существуют объекты,

которые присутствуют лишь во время вычисления выражения, но есть и такие как базы данных, которые существуют независимо от программы. Этот спектр сохраняемости объектов охватывает:

– промежуточные результаты вычисления выражений;

– локальные переменные в вызове процедур;

– собственные переменные (как в *ALGOL*-60), глобальные переменные и динамически создаваемые данные;

– данные, сохраняющиеся между сеансами выполнения программы;

– данные, сохраняемые при переходе на новую версию программы;

– данные, которые вообще переживают программу.

Традиционно первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Этот конфликт культур приводит к неожиданным решениям: программисты разрабатывают специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных переиначивают свою технологию под короткоживущие объекты.

Унификация принципов параллелизма для объектов позволила создать параллельные языки программирования. Аналогичным образом введение сохраняемости как нормальной составной части объектного подхода приводит нас к объектно-ориентированным базам данных (*OODB*, *object-oriented databases*). На практике подобные базы данных строятся на основе проверенных временем моделей – последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах объектов, время жизни которых превосходит время жизни отдельной программы. Как мы увидим в третьей главе, эта унификация значительно упрощает разработку отдельных видов приложений, позволяя, в частности, применить единый подход к разным сегментам программы, одни из которых связаны с базами данных, а другие не имеют такой связи.

Языки программирования, как правило, не поддерживают понятия сохраняемости; примечательным исключением является *Smalltalk*, в котором есть протоколы для сохранения объектов на диске и загрузки с диска. Однако записывать объекты в неструктурированные файлы – это все-таки наивный подход, пригодный только для небольших систем. Как правило, сохраняемость достигается применением (немногочисленных) коммерческих *OODB*. Другой вариант – создать объектно-

ориентированную оболочку для реляционных СУБД; это лучше, в частности, для тех, кто уже вложил средства в реляционную систему.

Сохраняемость – это не только проблема сохранения *данных*. В *OODB* имеет смысл сохранять и *классы*, так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти.

В заключение представим данное определение сохраняемости.

Сохраняемость – способность объекта существовать во времени, переживая породивший его процесс, и (*или*) *в пространстве, перемещаясь из своего первоначального адресного пространства.*

Преимущества объектной модели

Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Согласно нашему опыту, есть еще пять преимуществ, которые дает объектная модель.

Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Страуструп отмечает: «Не всегда очевидно, как в полной мере использовать преимущества такого языка, как *C++*. Существенно повысить эффективность и качество кода можно просто за счет использования *C++* в качестве «улучшенного *C*» с элементами абстракции данных. Однако гораздо более значительным достижением является введение иерархии классов в процессе проекти-

рования. Именно это называется *OOD* и именно здесь преимущества *C++* демонстрируются наилучшим образом» [8]. Опыт показал, что при использовании таких языков, как *Smalltalk*, *Object Pascal*, *C++*, *CLOS* и *Ada* вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно. *Сохраняемость поддерживает состояние и класс объекта в пространстве и во времени.*

Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, объектная модель уменьшает риск разработки сложных систем, прежде всего, потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель ориентирована на человеческое восприятие мира, или, по словам Робсона, «многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам» [8].

Использование объектного подхода

Возможность применения объектного подхода доказана для задач самого разного характера. В настоящее время объектно-ориентированное проектирование – единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако следует заметить, что иногда применение *OOD* может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки.

ГЛАВА 3. ВВЕДЕНИЕ В НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ

3.1. Непрерывная интеграция

Непрерывная интеграция – это практика разработки программного обеспечения, когда участники команды часто объединяют результаты своей работы друг с другом: от одного до нескольких раз в день. Каждая интеграция проверяется автоматической сборкой и тестированием для определения ошибок настолько быстро, насколько это возможно. Большинство команд находят, что этот подход существенно уменьшает количество проблем при интеграции и позволяет разрабатывать программы более быстро и качественно [10].

Этот подход достигается недорогими и сложными средствами, его сущность лежит в использовании простого правила для всей команды: хотя бы раз в день вносить сделанные изменения в репозиторий исходного кода.

Термин «непрерывная интеграция» (НИ) появился в процессе разработки под названием «Экстремальное программирование» как один из его оригинальных двенадцати практик.

Хотя практика непрерывной интеграции не требует особых средств, мы посчитали полезным использовать так называемый «сервер непрерывной интеграции». Наиболее известный в данный момент из таковых – *CruiseControl* – проект с открытыми исходными кодами, который начали разрабатывать несколько человек из *ThoughtWorks*, а теперь он поддерживается большим сообществом. Оригинальная версия была написана на *Java*, но также доступна на платформе *Microsoft* в виде *CruiseControl.NET*.

Разработка новой функциональности с непрерывной интеграцией. Самый простой способ объяснить, что такое НИ и как она работает – это описать небольшой пример разработки новой функции (*feature*). Давайте представим, что мне нужно сделать что-то с небольшой частью программной системы. На данный момент не имеет значение, в чем конкретно заключается задание, главное, что оно небольшое и может быть выполнено за несколько часов (более длительные задания и другие вопросы мы рассмотрим ниже).

Я начинаю с извлечения копии текущего исходного кода из репозитария на свой компьютер с помощью системы контроля версий, выполняя команду «*check out*».

Предыдущий параграф имеет смысл для людей, использующих систему контроля версий исходного кода, но не имеет смысла для всех остальных. Позвольте мне быстро описать, в чем ее суть. Система контроля версий хранит все исходные коды проекта в репозитории. Текущее состояние системы обычно характеризуется термином «*mainline*». В любое время разработчик может сделать контролируемую копию *mainline* на свою машину – это называется *checking out*. Копия разработчика называется *working copy* (рабочая копия). (В дальнейшем основной операцией будет обновление рабочей копии до *mainline* – на практике это почти то же самое.)

Итак, у меня есть рабочая копия исходных текстов системы, и я делаю все, чтобы выполнить задание. Точнее, выполнение задание заключается в изменении существующего работающего кода и написании автоматизированных тестов. Непрерывная интеграция подразумевает высокую степень покрытия системы автоматизированными тестами как средство самотестирования. Чаще всего, тесты построены на *XUnit*-фреймворках.

Как только работа выполнена (обычно даже во время работы), я выполняю автоматическую сборку на своей машине разработчика. Скрипт сборки берет исходный код из локальной копии, компилирует его, собирает в исполняемые модули и выполняет автоматизированные тесты. Сборка считается успешной, если компиляция прошла без ошибок и все тесты отработали правильно.

После успешной сборки можно подумать о том, чтобы сохранить (*commit*) сделанные изменения в репозиторий. Есть, конечно, загвоздка в том, что другие разработчики к этому моменту уже внесли свои изменения в репозиторий (чаще всего именно так и происходит). Я обновляю код из репозитария, собираю и тестирую проект еще раз, что может привести как к ошибке компиляции, так и к заваленным тестам. В этом случае исправление этих ошибок лежит в моей области ответственности. Этот процесс повторяется до тех пор, пока я не получаю полностью работающую сборку, синхронизированную с репозиторием, после чего можно записать свои изменения в репозиторий.

Однако на этом моя работа не заканчивается. В этот момент сборка проекта происходит снова, но на этот раз уже на интеграционном сервере на основе текущего состояния репозитария. Только когда сборка с моими изменениями прошла успешно на сервере, можно говорить о выполнении задания. Всегда есть вероятность того, что я что-то пропустил, и репозиторий содержит некорректный код, поэто-

му работа выполнена только в том случае, когда произошла сборка на независимом сервере с моими изменениями из репозитория. Эта интеграционная сборка может быть выполнена вручную, либо автоматически при помощи *CruiseControl*.

Если между двумя разработчиками происходит конфликт (например, одновременное изменение одних и тех же строк в одном файле), он обычно обнаруживается последним разработчиком во время сборки после обновления рабочей копии, либо во время интеграционной сборки на сервере. В любом случае ошибка быстро обнаруживается. В этом случае наиболее важным заданием является ее исправление, чтобы снова добиться успешной сборки. В среде непрерывной интеграции никогда нельзя держать интеграционную сборку в состоянии ошибки долгое время. В хорошей команде должно быть много успешных сборок в день; плохие сборки могут появляться время от времени, но должны быстро исправляться.

В результате такого подхода получается стабильный модуль системы, который работает надлежащим образом и содержит мало ошибок. Все разработчики используют этот стабильный код и никогда не уходят от него настолько далеко, чтобы интеграция заняла достаточно долгое время. Также меньше времени тратится на поиск ошибок, так как они быстро выявляются.

3.2. Развертывание программного обеспечения и непрерывная интеграция

Пример, описанный выше, является обзором НИ и описанием ее работы в реальной повседневной жизни. Плавное внедрение практики НИ в работу команды заключается не только в этом. Теперь давайте сфокусируемся на ключевых практиках, которые позволят эффективно использовать НИ.

Поддержка единого репозитория исходных кодов проекта. Проекты разработки информационных систем содержат большое количество файлов, которые требуют организации для сборки продукта. Отслеживание всех файлов требует больших усилий, особенно, если над ними работают одновременно несколько человек. На протяжении многих лет команды разработчиков сделали много систем для такого рода задач. Эти средства – системы управления исходным кодом, управление конфигурациями, репозиториями, системы контроля версий и множество других наименований – являются неотъемлемой частью большинства проектов.

Итак, вам стоит убедиться, что используется система контроля версий.

Хотя многие команды используют репозиторий, общей ошибкой является то, что они не кладут туда все файлы. Кроме исходного кода, там должны лежать тестовые скрипты, файлы настроек, схемы баз данных, установочные скрипты и сторонние библиотеки.

Случалось, что даже компилятор *C++* записывали в репозиторий (что было важно во времена разношерстных *C++* компиляторов). Основное правило заключается в том, чтобы вы смогли сесть за чистую машину, сделать *checkout* и полностью собрать и запустить проект. На этой машине должно быть минимальное количество необходимых разработчику вещей (обычно только большие, сложные для установки и стабильные). Например, операционная система, среда разработки *Java* или система управления базой данных (СУБД).

Вы должны положить в систему контроля версий все необходимое для сборки проектов, также вы можете использовать репозиторий для хранения других файлов, связанных с проектом. Файлы проектов *IDE* (среды разработки) обычно удобно хранить там, так как это наиболее простой способ разделить их с другими разработчиками.

Одна из функций системы контроля версий заключается в возможности создавать несколько веток (*branches*) для поддержки нескольких направлений разработки проекта. Эта полезная, даже обязательная функция часто используется слишком интенсивно и приводит к большим неудобствам для разработчиков. Используйте ветки по минимуму, в частности, используйте *mainstream* (обычно ее называют *trunk*) как единую ветку для разработки. Большую часть времени и работы должно выполняться в этой ветке (другие ветки обычно создаются для исправления ошибок перед развертыванием приложения в рабочем режиме (*production*) и для временных экспериментов).

В общем, вы должны хранить в репозитории все, что необходимо для сборки, но ничего, из того, что, собственно, собралось. Некоторые люди хранят результаты сборки также в репозитории, но я считаю, что это является индикатором более глубокой проблемы, обычно заключающейся в невозможности повторить сборку проекта.

Автоматизация сборки. Превращение исходных кодов системы в работающую систему обычно является сложным процессом, включающим компиляцию, перемещение файлов, загрузку схем в базы данных и т. п. Однако большинство операций в этой области разработки

ПО может быть автоматизировано, и, как следствие, должно быть автоматизировано. Требовать от людей кликать по диалоговым окнам – это потеря времени и потенциальная почва для ошибок.

Средства автоматической сборки – обычная часть среды разработки. Мир *Unix*-разработчиков десятилетиями пользовался утилитой *make*, сообщество *Java* разработало *Ant*, сообщество *.NET – NAnt* и в настоящее время имеет *MSBuild*. Убедитесь, что вы можете собирать и запускать систему, используя эти скрипты одной командой.

Общей ошибкой является то, что не все файлы включены в скрипт сборки. Сборка должна включать извлечение схемы базы данных из репозитория и загрузку ее в исполняемой среде. Я уточню сформулированное ранее правило: любой человек должен сесть за чистую машину, извлечь проект из репозитория, запустить одну команду и получить работающую систему.

Скрипты сборки могут быть различных видов, часто специализированные под платформу или сообщество, но не обязательно. Хотя большинство наших *Java*-проектов использует *Ant*, некоторые использовали *Ruby* (система *Ruby Rake* – очень приятное средство сборки). Также *Ant* нам очень пригодился для сборки проекта на *Microsoft COM*.

Полная сборка может занять довольно продолжительное время, хотя вы не должны выполнять все шаги, если были сделаны незначительные изменения. Хорошая система сборки должна анализировать сделанные изменения. Общепринятый способ – сравнивать даты изменения исходного файла и результата и выполнять операцию, только если исходный файл новее. Зависимости определяются сложнее – требуется определить, не изменился ли хотя бы один файл, который требуется для сборки модуля, если да – то пересобрать модуль. Компиляторы могут отслеживать, а могут и не отслеживать такого рода зависимости.

В зависимости от того, что вам нужно, вы можете выбирать, что именно вы хотите собрать. Вы можете собрать систему с тестами, либо без тестов, или с различными наборами тестов. Некоторые компоненты могут собираться отдельно. Скрипт сборки должен позволять запускать различные цели (*targets*) для различных случаев.

Большинство из нас используют *IDE* (среды разработки), и большинство сред имеют какую-либо встроенную систему сборки. Однако эти системы обычно специфичны для конкретной среды и часто ограничены по функциональности. Кроме того, они требуют, чтобы среда разработки была запущена. Это нормально для пользователей, рабо-

тающих в *IDE* и использующих ее для разработки. Однако, очевидно, что должен быть основной скрипт, который может быть запущен на сервере, а также была возможность запустить его из другого скрипта. Таким образом, мы не против, что *Java* разработчики собирают проект с помощью среды разработки, но основной скрипт сборки использует *Ant*, чтобы гарантировать то, что проект может быть собран на сервере.

Сделайте свою сборку самотестируемой. Традиционно сборка подразумевает компиляцию, линковку (связывание) и все остальное, чтобы программа запустилась. Программа может запуститься, но это не означает, что она будет делать то, что нужно. Современные сильно типизированные языки программирования могут выявить много ошибок, но пропустят значительно больше.

Хорошим способом для обнаружения ошибок может служить включение процесса тестирование кода в процесс сборки. Конечно, это тестирование не совершенно, но оно может отловить достаточное количество ошибок, чтобы оправдать свое существование. Широкое распространение практик экстремального программирования (*XP*) и разработки через тестирование (*TDD*) особенно повлияло на популяризацию концепции самотестирующегося кода, которая была по достоинству оценена многими людьми.

Я большой поклонник и *XP* и *TDD*, но ни один из этих подходов не обязателен для того, чтобы получить все преимущества от самотестирования кода. Оба этих подхода предполагают написание тестов для кода до разработки самого кода – в этом режиме тесты нужны для разработки архитектуры системы, в тоже время их предназначение – обнаружение ошибок. Это отличная практика, но не обязательна для целей непрерывной интеграции, где у нас более слабые требования к самотестируемому коду. (Хотя я предпочитаю *TDD* для создания самотестируемого кода.)

Для самотестируемого кода вам необходим набор тестов, покрывающих большую часть рабочего кода. Тесты должны запускаться одной простой командой и проверять самих себя. Результатом запуска тестов должен являться факт того, прошли ли проверку все тесты, или хотя бы один не прошел. Ошибка хотя бы одного теста должна означать ошибку всей сборки.

За последние несколько лет подъема *TDD* сделало популярными семейство фреймворков с открытыми исходными кодами *XUnit*, идеальными для такого рода тестирования.

Средства *XUnit*, конечно, являются отправной точкой для разработки тестов, но также следует обратить внимание на другие средства, позволяющие разрабатывать более широкий спектр тестов. В настоящее время существует довольно большой выбор, например, *FIT*, *Selenium*, *Sahi*, *Warir*, *FITnesse* и множество других, для которых я не буду приводить полный перечень.

Конечно, вы не можете рассчитывать, что тесты полностью избавят вас от ошибок. Как часто говорят: тесты не доказывают отсутствие ошибок. Однако стремление к совершенству – не единственная причина, по которой вы пишите тесты. Несовершенные, но часто выполняющиеся тесты значительно лучше совершенных, но никогда не написанных.

Все сохраняют изменения в репозитории каждый день. Интеграция, в первую очередь, связана с коммуникациями. Интеграция позволяет разработчикам сообщить другим разработчикам о своих изменениях. Частые коммуникации позволяют людям узнавать о новых изменениях, как только они появились.

Единственное условие для разработчика, сохраняющего изменения в репозиторий (*commit*), заключается в том, что его код корректно собирается, что, конечно, включает успешное выполнение тестов. Как и для любого цикла фиксации изменений, разработчик вначале обновляет свою рабочую копию из репозитория, разрешает все конфликты, затем собирает проект на своей локальной машине. Если сборка проходит успешно, изменения можно отправлять в репозиторий.

Часто выполняя эту процедуру, разработчики быстро обнаруживают конфликты между собой. Ключ к быстрому решению проблем – в их быстром обнаружении. Если разработчики заносят свои изменения каждые несколько часов, конфликт обнаруживается быстро, и также просто разрешается, так как к этому моменту версии кода еще не сильно разошлись. Конфликты, которые не обнаруживаются в течение недель, разрешаются очень сложно.

Фактически, то, что вы собираете проект после обновления рабочей копии, означает, что вместе с текстовыми ошибками вы также определяете ошибки компиляции. Если код покрыт тестами, вы также обнаруживаете ошибки в работе кода. Последний вид конфликтов позволяет избавиться от особенно трудных для обнаружения ошибок, которые могут оставаться незамеченными долгое время. Так как между изменениями проходит всего лишь несколько часов работы, мест, где могут быть проблемы, не так много. К тому же, так как самих из-

менений не так много, вы можете использовать отладку по разнице между файлами (*diff*) для определения места ошибки.

Основное правило заключается в том, что каждый разработчик должен фиксировать свои изменения в репозитории каждый день. На практике даже чаще полезней делать это несколько раз в день. Чем чаще вы фиксируете свои изменения, тем меньше у вас будет конфликтов и проще будет разрешить те, что есть.

Частая фиксация изменений способствует тому, чтобы разработчики делили свою работу на маленькие подзадачи по нескольку часов каждая. Это помогает отслеживать прогресс и придает ощущение прогресса в работе. Часто сначала люди чувствуют, что не могут сделать что-либо значительное всего за несколько часов работы, но мы выяснили, что этому учит хороший куратор и практика.

Каждое изменение в репозитории должно включаться в сборку на интеграционном сервере. *При* ежедневной фиксации изменений достигается частое выполнение тестов в проекте. Это означает то, что код в репозитории находится в жизнеспособном состоянии. Однако на практике дела могут обстоять не так хорошо. Одна из причин заключается в том, что разработчики не делают обновление и сборку перед фиксацией изменений. Источником другой являются различия в настройках окружения в компьютерах разработчиков.

В результате вы должны быть уверены, что сборки производятся регулярно на интеграционном сервере и изменение считается сделанным только в том случае, если сборка успешно прошла на этом сервере. Так как за изменение отвечает сделавший его разработчик, этому человеку нужно отслеживать сборку на сервере, чтобы он мог исправить ее, если она пройдет неуспешно. Из этого можно сделать вывод, что вы не должны уходить домой, пока не пройдет сборка со всеми изменениями, которые вы сделали за день.

Чтобы этого добиться, существуют 2 пути: использовать ручную сборку, либо сервер непрерывной интеграции.

Подход с ручной сборкой описывается наиболее просто. Очевидно, это примерно то же самое, что и сборка на локальной машине, которую разработчик делает перед фиксацией изменений. Разработчик идет к интеграционному серверу, обновляет там рабочую копию (в которую попадают его последние изменения из репозитория) и запускает интеграционную сборку. Он следит за процессом сборки, и как только сборка прошла успешно – его работа выполнена.

Сервер непрерывной интеграции играет роль монитора репозитория. Каждый раз, как только произошла фиксация изменений, сервер автоматически извлекает эти изменения, запускает сборку и оповещает автора изменений о результатах сборки. Автор изменений не уходит, пока не получит оповещения – обычно по электронной почте.

Чтобы подход НИ эффективно работал – должны использоваться все необходимые практики. Но на самом деле многие команды, успешно работающие по НИ, находят *CruiseControl* полезным средством.

Часто в проекте выполняются регулярные сборки по расписанию, например, каждую ночь. Это не то же самое, что непрерывная сборка и этого недостаточно для непрерывной интеграции. Основной целью НИ является обнаружение проблем настолько быстро, насколько это возможно. Ночные сборки (*nightly builds*) означают, что ошибки останутся невыявленными целый день, пока кто-нибудь их не обнаружит. Так как они находятся в системе продолжительное время, их поиск и устранение также займет большее время.

Ключевым моментом в непрерывной интеграции является то, что если сборка прошла неуспешно, она должна быть немедленно исправлена. Аксиомой при работе с НИ является то, что вы всегда работаете со стабильным кодом. Ничего страшного в том, что сборка на сервере все-таки не прошла, хотя если это случается все время, стоит рекомендовать людям быть более внимательными при обновлении и локальной сборке перед фиксацией изменений. Однако, когда сборка на сервере все же провалилась, очень важно исправить это как можно быстрее.

Обычно, когда команда знакомится с НИ, это наиболее сложная вещь для понимания. Вначале в команде идет борьба за то, чтобы выработать привычку держать код в репозитории в рабочем состоянии, особенно, если они работают с уже существующим кодом. Терпение и настойчивость в применении этого правила, кажется, всегда приводила к результату.

Делайте сборку быстрой. Основная цель непрерывной интеграции – обеспечить быструю обратную связь. Ничто так не подрывает идею НИ, как долго выполняющаяся сборка.

Однако для большинства проектов рекомендации *XP* десятиминутной сборки подходят идеально. Большинство наших новых проектов достигают этой цели. Это действительно стоит того, потому что каждая минута, которую вы выигрываете в процессе сборки, сохраня-

ет минуту для каждого разработчика каждый раз перед фиксацией изменений. Так как практика НИ требует частых фиксаций, это экономит много времени.

Если вы наблюдаете за сборкой, продолжающейся в течение часа, работа по ее оптимизации может показаться «страшной» перспективой. Такие проблемы могут возникнуть даже при работе с новым проектом. Для корпоративных приложений, как минимум, обычно узким местом является тестирование, особенно те из тестов, которые используют внешние сервисы, например, базу данных.

Возможно, ключевым моментом здесь является настройка поэтапной сборки. Идея поэтапной сборки (также известной как «последовательная сборка» – *build pipeline*) заключается в том, что несколько сборок выполняется последовательно. Фиксация изменений вызывает первую сборку (*commit build*). Она должна выполняться быстро и поэтому мы должны ее сократить, что уменьшит ее способность обнаруживать ошибки. «Трюк» заключается в следующем: соблюдать баланс между требованиями по обнаружению ошибок и скоростью, чтобы первая сборка (*commit build*) была достаточна для продолжения работы.

Как только первая сборка будет достаточно надежна, чтобы люди могли уверенно работать с кодом дальше, можно запустить следующую, которая будет выполняться значительно дольше и включать тестирование, не вошедшее в первую сборку. Для этого могут использоваться дополнительные серверы, так как тестирование занимает значительно больше времени.

В качестве простого примера возьмем двухэтапную сборку. На первом этапе производится компиляция и запускаются тесты, которые работают только локально и не используют базу данных. Такие тесты работают очень быстро, удерживая общее время сборки в рамках 10 мин. Однако ошибки интеграции между модулями системы, особенно ошибки, связанные с реальной базой данных, не будут обнаружены. Сборка второго этапа запускает различные наборы тестов, работающие с базой данных и проверяющие больше интеграционные аспекты системы. Второй этап может занимать пару часов.

При таком сценарии разработчики используют первый этап в основном цикле НИ. Сборка второго этапа запускается, когда это возможно, используя результаты последней успешной сборки первого этапа для прогонки тестов. Если второй этап не собирается, это не означает, что вся работа останавливается, но команда должна добиться

исправления обнаруженных ошибок как можно скорее, сохраняя сборку первого этапа в жизнеспособном состоянии. Несмотря на то, что сборка второго этапа не обязана всегда быть успешной, каждая обнаруженная ей ошибка должна быть исправлена как минимум в течение нескольких следующих дней.

Если сборка второго этапа обнаруживает ошибку – это означает, что первая сборка должна включать еще один тест. Настолько, насколько это возможно, вы должны быть уверены в том, что провал второй сборки приводит к появлению новых тестов в первой сборке, чтобы ошибка «отлавливалась» на первом этапе. Этот путь приводит к увеличению объема тестирования при первой сборке. Однако существуют случаи, когда невозможно использовать быстро выполняющийся тест для проверки такой ошибки, в этом случае вы можете решить оставить этот тест в сборке второго этапа. К счастью, в большинстве случаев этого можно избежать.

Этот пример демонстрирует двухэтапную сборку, но описанные выше основные принципы могут использоваться в схемах с любым количеством этапов. Сборки после первой могут запускаться параллельно, например, если у вас второй этап длится два часа, вы можете уменьшить время отклика сборки, разделив объем тестирования поровну на два компьютера. Используя вторичные параллельные сборки, вы можете включить дополнительные виды автоматизированного тестирования, такие как тестирование производительности, в свой обычный процесс.

Производите тестирование в копии среды реальной эксплуатации. Цель тестирования – выявить любую проблему, которая может проявиться в среде эксплуатации (*production environment*) в контролируемых условиях. Значительную роль в этом играет среда выполнения, которая используется в *production*. В случае тестирования в другой среде каждое отличие оборачивается риском того, что то, что случится в среде тестирования, никогда не проявится в среде эксплуатации.

В результате вы хотите настроить среду тестирования настолько ближе к среде работы приложения, насколько это возможно. Используйте те же самые СУБД, тех же версий, ту же версию операционной системы. Положите все необходимые библиотеки из рабочей среды в тестовую, даже если система их не использует. Используйте те же *IP*-адрес и порты, запускайте систему на том же оборудовании.

Однако в реальности существуют ограничения. Если вы пишите настольное ПО, практически нереально его протестировать на каждом возможном настольном компьютере со всеми вариантами установленного ПО других фирм, которое может использоваться. Также среда эксплуатации системы может быть неоправданно дорогой для клонирования в тестовых целях (хотя я часто наблюдал случаи «ложной» экономии для систем средней стоимости). Несмотря на все эти ограничения, вашей целью должно стать копирование среды эксплуатации настолько точно, насколько вы можете, и понимание рисков тех отличий, которые все же остались.

Если у вас достаточно простой автоматизированный процесс установки, без большого количества неудобных коммуникаций, есть возможность запускать сборку после фиксации изменений в копии среды эксплуатации. Однако часто придется использовать двухэтапную сборку, так как интеграция с внешними системами может быть медленной и ненадежной. В результате общепринято иметь искусственную среду для быстрой первичной сборки и копию среды эксплуатации для вторичного тестирования.

Я замечаю растущий интерес к использованию виртуализации для упрощения развертывания тестовой среды. Виртуальные машины могут быть сохранены со всеми необходимыми элементами, заложенными в технологии виртуализации. Довольно просто установить последнюю сборку и запустить тестирование. Кроме этого есть возможность запускать несколько пакетов тестов на одной машине, или имитировать несколько машин в одной сети на одной физической машине. Использование виртуальных машин приобретает все больше смысла с увеличением их производительности.

Простой доступ к файлам последней сборки. Один из наиболее сложных моментов в разработке программного обеспечения заключается в том, чтобы быть уверенным, что вы делаете именно то, что нужно пользователям. Очень трудно заранее специфицировать, что именно нужно сделать и не ошибиться в этом; часто людям удобнее посмотреть на то, что не совсем правильно, и сказать, что конкретно нужно исправить. Гибкие (*Agile*) процессы разработки явно рассчитывают на эту особенность человеческого поведения.

Для того чтобы этот принцип работал, все участники проекта должны иметь возможность получать исполняемые файлы последней сборки и запускать их для демонстрационных целей, тестирования, или просто посмотреть, что изменилось за последнюю неделю.

Сделать это достаточно просто: нужно убедиться, что есть место, где лежат последняя сборка, и о котором все знают. Также может быть полезным выкладывать туда несколько последних сборок. Для самой последней сборки вы должны быть уверены в том, что она прошла тестирование на достаточно «сильном» наборе тестов.

Если вы следуете процессу с хорошо определенными итерациями, можно выкладывать сборки также по окончании итерации. Обычно при демонстрациях продукта нужна версия, с которой знаком докладчик, поэтому обычно стоит пожертвовать последними наработками, чтобы успешно провести презентацию на знакомой и стабильной версии.

Все должны знать, что происходит. Как уже упоминалось, НИ тесно связана с коммуникациями в проекте. Вы должны быть уверены в том, что все знают, где посмотреть текущее состояние системы и изменения, которые в ней появились.

Одной из наиболее важных вещей является состояние основной сборки. Если вы используете *CruiseControl*, то вам доступен *web*-сайт, на котором видно, собирается ли проект в данный момент и каков статус последней сборки. Во многих командах любят использовать более привлекательные средства для непрерывной индикации состояния сборки, например, зеленая лампочка, когда сборка проходит, и красная – если сборка не прошла. Особенно часто используются красные и зеленые лавовые лампы, которые не только отображают, что сборка не прошла, но и время, которое она была в этом состоянии. По пузырькам на красной лампе можно определить, что сборка была сломана слишком долго.

Визуальная составляющая необходима, даже если вы используете ручной процесс непрерывной интеграции. Монитор компьютера, на котором производится сборка, может показывать состояние основной сборки. Часто можно использовать некоторый предмет на столе того, кто производит сборку (опять же это может быть что-нибудь «дурацкое», вроде резинового цыпленка). Часто люди любят, чтобы при нормальной сборке издавался какой-то звук, вроде звонка.

Конечно, веб-страница на сервере непрерывной интеграции может содержать значительно больше полезной информации. *CruiseControl* отображает информацию не только о том, что в данный момент собирается, но и какие изменения были сделаны. *Cruise* также хранит историю изменений, позволяя команде следить за недавними активностями в проекте.

Другое преимущество в использовании веб-сайта есть для команд, которые работают удаленно, чтобы получать актуальный статус проек-

та. В общем, я предпочитаю, чтобы люди, которые активно работают на проекте, сидели вместе, но часто есть удаленные участники, которые тоже хотят быть в курсе. Для групп разработчиков также полезно агрегировать информацию о статусе сборок сразу нескольких проектов.

Хорошим информационным дисплеем может быть не только экран компьютера. Приведу один пример из проекта, в который внедрялась непрерывная интеграция. На протяжении долгого времени не удавалось делать стабильные сборки. Тогда мы повесили на стену календарь на целый год и каждый день *QA*-группа отмечала зеленым стикером дни, когда сборка проходила основные тесты, и красным – если нет. Со временем календарь показал устойчивое улучшение процесса сборки, когда зеленые квадраты стали обычным явлением – календарь сняли. Его задача была выполнена.

Автоматическое развертывание. Чтобы использовать непрерывную интеграцию, вам необходимы несколько сред выполнения (*environment*): одна, чтобы исполнять основной набор тестов (*commit tests*) и одна или более – для запуска вторичных тестов. Так как вы перемещаете исполняемые файлы между этими средами по нескольку раз в день, конечно, вы захотите делать это автоматически. Таким образом, важно иметь скрипты, которые позволят вам просто разворачивать приложение в любую среду выполнения.

Как следствие, вы сможете написать скрипты для развертывания приложения для реальной эксплуатации (*production*) так же просто. Возможно, вы не будете обновлять *production* каждый день (хотя я работал в таких проектах), но автоматическое развертывание поможет как ускорить процесс, так и уменьшить количество ошибок. Потом вы все равно получаете эту возможность почти бесплатно, так как, скорее всего, будут использоваться те же средства, что и для развертывания в среде тестирования.

Если вы автоматически развертываете приложение в среде эксплуатации, нужно также предусмотреть и автоматический откат. Время от времени случается что-то «ужасное», поэтому нужно иметь возможность быстро вернуться к последнему стабильному состоянию. Это также уменьшит напряжение при внедрении новой сборки, разработчики не будут бояться чаще развертывать новую сборку и, следовательно, чаще поставлять пользователям новые функции. (Сообщество *ruby on rails* разработало утилиту *Capistrano*, которая является хорошим примером такого рода вещей.)

Миграция базы данных является общим препятствием для многих людей, которые выпускают частые релизы системы. Изменять базу данных неудобно, потому что недостаточно всего лишь изменить ее схему, намного сложнее убедится, что данные тоже были перенесены корректно. С одним интересным вариантом автоматического развертывания я столкнулся при разработке одного общедоступного *web*-приложения. Идея заключалась в том, чтобы развертывать временную версию для некоторого подмножества пользователей. Это позволяет протестировать новые функции и пользовательский интерфейс до финального релиза. Автоматическое развертывание является частью дисциплины непрерывной интеграции и необходимо для полноценной работы.

Преимущества непрерывной интеграции. В целом самым большим преимуществом непрерывной интеграции я считаю снижение уровня рисков.

Проблема с отложенной интеграцией заключается в том, что трудно предсказать, сколько времени она займет, и, что еще хуже, очень сложно определить, сколько же еще осталось до конца. В результате вы чувствуете себя «слепым котенком» в одном из наиболее напряженных этапов проекта, даже если (в редких случаях) вы еще укладываетесь в сроки.

Непрерывная интеграция полностью устраняет эту проблему. Продолжительная интеграция отсутствует, вы всегда контролируете ситуацию. В любое время вы знаете, где вы, что работает, что не работает и какие критические ошибки есть в вашей системе.

Ошибки – это «отвратительные вещи», которые разрушают уверенность, и ломают расписания и репутации. Ошибки в развернутых приложениях злят пользователей, и больше всего достается вам. Ошибки, портящие работу в ее середине, делают всю систему неработоспособной.

Непрерывная интеграция не избавляет от ошибок, но делает их поиск и устранение существенно проще. В каком-то отношении это похоже на самотестирующийся код. Если вы сделали ошибку и быстро ее нашли, избавится от нее намного проще. Так как вы изменили только небольшую часть системы, вам не придется далеко смотреть. К тому же вы только что там работали, поэтому не придется много освежать в памяти – опять же ошибка исправится быстрее. Всегда можно посмотреть разницу между кодом системы с ошибкой и без нее.

Ошибки обычно накапливаются. Чем больше ошибок, тем сложнее избавится от каждой. Частично из-за того, что они могут взаимодействовать друг с другом (когда симптом ошибки проявляется как следствие нескольких ошибок в коде, найти каждую значительно труднее). Есть и психологический фактор – сложнее находить и исправлять ошибки, когда их много – феномен, которые Pragmatic Programmers называют синдромом «Разбитых окон» («Broken Windows»).

В результате проекты с непрерывной интеграцией порождают значительно меньше ошибок, как при эксплуатации, так и при разработке. Однако стоит предупредить, что степень этого преимущества прямо зависит от того, насколько хорош ваш набор тестов. Вы найдете, что не слишком сложно составить набор тестов, чтобы добиться существенных результатов. Впрочем, обычно проходит некоторое время до того, как команда реально добирается до настолько низкого количества ошибок, которого потенциально может добиться. Это достигается постоянной разработкой новых и улучшением старых тестов.

Непрерывная интеграция устраняет один из наибольших барьеров к частому выпуску релизов. Это важно, потому что позволяет пользователям сразу начинать пользоваться новыми функциями, разработчикам – быстро получать отзывы на эти функции и в целом процесс разработки становится более интерактивным. Это позволяет «разбить стену» между заказчиками и разработчиками, которая, я верю, является самым большим препятствием к успешной разработке программного обеспечения.

Знакомясь с непрерывной интеграцией. Итак, вам хочется попробовать непрерывную интеграцию – с чего можно начать? Полный набор практик, описанный выше, даст вам все преимущества подхода, но совершенно не обязательно использовать их все сразу.

Здесь нет единого рецепта на все случаи – все зависит от ваших установок и команды. Но есть необходимые условия, чтобы все заработало.

Один из первых шагов – автоматизация сборки. Положите все необходимое в репозиторий исходных кодов и добейтесь того, чтобы сборка производилась одной командой. Для многих проектов это уже большое дело – даже этого достаточно, чтобы заработали многие другие практики. В начале вы можете делать сборку даже вручную, по требованию или автоматизировать сборку по ночам. Хотя это еще не непрерывная интеграция, автоматизированная ночная сборка – отличное начало.

Добавьте некоторые автоматизированные тесты в вашу сборку. Попробуйте определить основные области, где могут быть сбои, и напишите для них автоматизированные тесты для быстрого выявления этих ошибок. Особенно трудно хорошо покрыть тестами уже существующий проект, поэтому надо с чего-то начинать.

Попробуйте ускорить сборку после изменений. Непрерывная интеграция со сборкой в несколько часов лучше, чем ничего, но магическая цифра – 10 мин – значительно лучше. Обычно это требует довольно серьезного хирургического вмешательства в ваш код, во время того, как вы разрываете зависимости между медленными участками системы.

Если стартует ваш новый проект, начните сразу с непрерывной интеграции. Следите за временем сборки и принимайте меры, как только вы выходите за порог в 10 мин. Быстрые действия позволят вам вовремя реструктуризировать код до того, как он вырастет настолько, что станет большой проблемой.

ГЛАВА 4. МЕЖПЛАТФОРМЕННЫЕ СИСТЕМЫ СБОРКИ И УПРАВЛЕНИЯ ПРОЕКТАМИ

4.1. Работа с *Maven*

Maven – это средство для управления и сборки проектов. Он позволяет разработчикам полностью управлять жизненным циклом проекта. Благодаря этому команда может автоматизировать процессы, связанные со сборкой, тестированием, упаковкой проекта и т. д.

В случае, когда над проектом работает несколько команд разработчиков, *Maven* позволяет работать в соответствии с определенными стандартами.

Вот основные аспекты, которыми позволяет управлять *Maven*:

– создание;
– документирование;
– отчеты;
– зависимости;
– релизы;
– *SCM*;
– список рассылки;
– дистрибьюция.

Основной целью *Maven* является предоставление разработчику:

– понятной модели для проектов, которая может быть использована повторно и была бы проста в поддержании;

– плагинов, которые могут взаимодействовать с этой моделью.

Структура и содержание проекта *Maven* указывается в специальном *xml*-файле, называемом *Project Object Model* (*POM*), который является базовым модулем всей системы.

Соглашение по конфигурации. В *Maven* используется соглашение по конфигурации.

Разработчики не обязаны указывать каждую деталь. *Maven* обеспечивает поведение проекта по умолчанию. Когда мы создаем проект с использованием этой технологии, то *Maven* создает базовую структуру проекта. Разработчики несут ответственность только за соответственное размещение файлов.

Ниже указаны значения по умолчанию для исходного кода проекта и других модулей (табл. 4.1).

Таблица 4.1

Значения по умолчанию для исходного кода проекта и других модулей *Maven*

Исходный код	***/src/main/java***
Ресурсы	*/src/main/resources*
Тесты	*/src/test*
Дистрибутив *JAR*	*/target*
Скомпилированный байт-код	*/target/classes*

Для того чтобы построить проект, *Maven* предоставляет нам настройки для управления жизненным циклом проекта и его зависимостями. Большая часть его задач поддерживается плагинами *Maven*.

POM

POM (*Project Object Model*) является базовым модулем *Maven*. Это специальный *XML*-файл, который всегда хранится в базовой директории проекта и называется ***pom.xml***.

Файл *POM* содержит информацию о проекте и различных деталях конфигурации, которые используются *Maven* для создания проекта.

Этот файл также содержит задачи и плагины. Во время выполнения задач *Maven* ищет файл *pom.xml* в базовой директории проекта. Он читает его и получает необходимую информацию, после чего выполняет задачи.

Среди конфигураций *Maven* мы можем выделить следующие:

– зависимости проекта;

– плагины;

– задачи;

– профиль создания;

– версия проекта;

– разработчики;

– список рассылки.

Перед тем как создавать ***pom.xml***, нам необходимо, прежде всего, определить группу проекта (*groupId*), его имя (*artifactId*) и его версию. Все это поможет нам унифицировать проект для простой его идентификации в репозитории.

Рассмотрим простейший пример ***pom.xml*** файла:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
 </project>
```

Каждый проект должен иметь свой собственный *POM*-файл.

Все *POM*-файлы должны иметь три обязательных элемента: ***groupId***, ***artifactId***, ***version***.

В репозитории проект выглядит следующим образом: ***groupId*:*artifactId*:*version***.

Ключевой элемент в *POM*-файле – это ***project***, который делится на три главных подгруппы:

– ***groupId***. Это *ID* группы проекта. Зачастую это уникальная организация или проект. Например, если мы хотим создать группу, которая отвечает за видео, то *groupId* будет выглядеть примерно так: *net.proselyte.video*. В этой группе будут все проекты, которые относятся к видео;

– ***artifactId***. Это идентификатор самого проекта. Чаще всего – его имя. Например, *maven-video*. *artifactId* также помогает найти проект в репозитории;

– ***version***. Версия проекта. Определяет конкретную версию продукта.

Супер *POM*

Все *POM*-файлы являются наследниками родительского *POMa*. Этот *POM*-файл называется ***Super POM*** и содержит значения, унаследованные по умолчанию.

Простой способ просмотреть настройки по умолчанию *Super POM* файла – это использование команды:

mvn help:effective-pom

Для понимания того, как это работает на практике, рассмотрим простой пример.

Создадим простой *Maven* проект. Открываем консоль и пишем следующее:

mvn help:effective-pom

В результате мы получим примерно следующий результат:

[INFO] Scanning for projects...
Downloading:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar-plugin/2.2/maven-jar-plugin-2.2.pom
Downloaded:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar-plugin/2.2/maven-jar-plugin-2.2.pom (9 KB at 15.5 KB/sec)
Downloading:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/10/maven-plugins-10.pom
Downloaded:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/10/maven-plugins-10.pom (8 KB at 57.8 KB/sec)
Downloading:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar-plugin/2.2/maven-jar-plugin-2.2.jar
Downloaded:
http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-jar-plugin/2.2/maven-jar-plugin-2.2.jar (27 KB at 184.5 KB/sec)
[INFO]
[INFO] --
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] --
[INFO]

[INFO] --- maven-help-plugin:2.2:effective-pom (default-cli) @ MavenTutorial ---
[INFO]

Зададим файл конфигурации:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>ProselyteTutorials</groupId>
<artifactId>MavenTutorial</artifactId>
<version>1.0-SNAPSHOT</version>
<repositories>
<repository>
<snapshots>
<enabled>false</enabled>
</snapshots>
<id>central</id>
<name>Central Repository</name>
<url>http://repo.maven.apache.org/maven2</url>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<releases>
<updatePolicy>never</updatePolicy>
</releases>
<snapshots>
<enabled>false</enabled>
</snapshots>
<id>central</id>
<name>Central Repository</name>
<url>http://repo.maven.apache.org/maven2</url>
</pluginRepository>
</pluginRepositories>
<build>
<sourceDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/src
/main/java</sourceDirectory>
```

<scriptSourceDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/src
/main/scripts</scriptSourceDirectory>
<testSourceDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/src
/test/java</testSourceDirectory>
<outputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/classes</outputDirectory>
<testOutputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/test-classes</testOutputDirectory>
<resources>
<resource>
<direc-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/src
/main/resources</directory>
</resource>
</resources>
<testResources>
<testResource>
<direc-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/src
/test/resources</directory>
</testResource>
</testResources>
<direc-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get</directory>
<finalName>MavenTutorial-1.0-SNAPSHOT</finalName>
<pluginManagement>
<plugins>
<plugin>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.3</version>
</plugin>
<plugin>
<artifactId>maven-assembly-plugin</artifactId>
<version>2.2-beta-5</version>

```
</plugin>
<plugin>
<artifactId>maven-dependency-plugin</artifactId>
<version>2.1</version>
</plugin>
<plugin>
<artifactId>maven-release-plugin</artifactId>
<version>2.0</version>
</plugin>
</plugins>
</pluginManagement>
<plugins>
<plugin>
<artifactId>maven-clean-plugin</artifactId>
<version>2.5</version>
<executions>
<execution>
<id>default-clean</id>
<phase>clean</phase>
<goals>
<goal>clean</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<artifactId>maven-resources-plugin</artifactId>
<version>2.3</version>
<executions>
<execution>
<id>default-testResources</id>
<phase>process-test-resources</phase>
<goals>
<goal>testResources</goal>
</goals>
</execution>
<execution>
<id>default-resources</id>
<phase>process-resources</phase>
```

```
<goals>
<goal>resources</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<artifactId>maven-jar-plugin</artifactId>
<version>2.2</version>
<executions>
<execution>
<id>default-jar</id>
<phase>package</phase>
<goals>
<goal>jar</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<version>2.0.2</version>
<executions>
<execution>
<id>default-compile</id>
<phase>compile</phase>
<goals>
<goal>compile</goal>
</goals>
</execution>
<execution>
<id>default-testCompile</id>
<phase>test-compile</phase>
<goals>
<goal>testCompile</goal>
</goals>
</execution>
</executions>
</plugin>
```

```
<plugin>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.10</version>
<executions>
<execution>
<id>default-test</id>
<phase>test</phase>
<goals>
<goal>test</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<artifactId>maven-install-plugin</artifactId>
<version>2.3</version>
<executions>
<execution>
<id>default-install</id>
<phase>install</phase>
<goals>
<goal>install</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<artifactId>maven-deploy-plugin</artifactId>
<version>2.7</version>
<executions>
<execution>
<id>default-deploy</id>
<phase>deploy</phase>
<goals>
<goal>deploy</goal>
</goals>
</execution>
</executions>
</plugin>
```

```
<plugin>
<artifactId>maven-site-plugin</artifactId>
<version>3.0</version>
<executions>
<execution>
<id>default-site</id>
<phase>site</phase>
<goals>
<goal>site</goal>
</goals>
<configuration>
<outputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/site</outputDirectory>
<reportPlugins>
<reportPlugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
</reportPlugin>
</reportPlugins>
</configuration>
</execution>
<execution>
<id>default-deploy</id>
<phase>site-deploy</phase>
<goals>
<goal>deploy</goal>
</goals>
<configuration>
<outputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/site</outputDirectory>
<reportPlugins>
<reportPlugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
</reportPlugin>
</reportPlugins>
</configuration>
```

</execution>
</executions>
<configuration>
<outputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/site</outputDirectory>
<reportPlugins>
<reportPlugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
</reportPlugin>
</reportPlugins>
</configuration>
</plugin>
</plugins>
</build>
<reporting>
<outputDirec-
tory>/home/proselyte/Programming/Projects/Proselyte/MavenTutorial/tar
get/site</outputDirectory>
</reporting>
</project>

Результат выполнения:

[INFO] --
[INFO] BUILD SUCCESS
[INFO] --
[INFO] Total time: 2.158s
[INFO] Finished at: Sun Mar 27 20:48:48 EEST 2021
[INFO] Final Memory: 11M/150M
[INFO] --

В полученном файле мы можем видеть изначальную структуру проекта, директорию вывода, необходимые плагины, репозитории, которые *Maven* будет использовать во время выполнения необходимых задач.

Maven также обеспечивает множество архитипов для создания проектов для создания определенной структуры и файла *pom.xml*.

4.2. Сборка проекта *Maven*

Жизненный цикл сборки

Жизненный цикл сборки в *Maven* – это четко определенная последовательность фаз, во время выполнения которых должны быть достигнуты определенные цели.

Типичный жизненный цикл сборки *Maven* представлен в табл. 4.2.

Таблица 4.2

Жизненный цикл сборки Maven

Фаза	Действия	Описание
prepare-resources	Копирование ресурсов	В этой фазе происходит копирование ресурсов, которое также может быть настроено
compile	Компиляция	В этой фазе происходит компиляция исходного кода
package	Создание пакета	В этой фазе в зависимости от настроек создается архив JAR/WAR. Тип архива указывается в pom.xml файле
install	Установка	В этой фазе происходит установка пакета в локальный/удаленный репозиторий maven

Каждая из этих фаз имеет фазы **pre** и **post.** Они могут быть использованы для регистрации задач, которые должны быть запущены перед и после указанной фазы.

Когда Maven начинает сборку проекта, он проходит через определенную последовательность фаз и выполняет определенные задачи, которые указаны в каждой из фаз. *Maven* имеет 3 стандартных жизненных цикла:

1) clean;
2) default;
3) site.

Задача (goal) – это специальная задача, которая относится к сборке проекта и его управлению. Она может привязываться как к нескольким фазам, так и ни к одной. Задача, которая не привязана ни к одной фазе, может быть запущена вне фаз сборки с помощью прямого вызова.

Порядок выполнения зависит от порядка вызова целей и фаз.

Например, рассмотрим команду ниже. Аргументы *clean* и *package* являются фазами сборки до тех пор, пока *dependency*: *copy-dependencies* является задачей.

mvn clean dependency:copy-dependencies package

В этом случае сначала будет выполнена фаза *clean*, после этого будет выполнена задача *dependency*: *copy-dependencies*. После чего будет выполнена фаза *package*.

Жизенный цикл *Clean*

Когда вы выполняете команду:

mvn post-clean

задача *clean Maven (clean:clean*) привязывается к фазе *clean* в жизненном цикле сборки. Эта задача удаляет ввод сборки путем удаления директории сборки. Таким образом, когда выполняется команда ***mvn clean***, *Maven* удаляет директорию сборки.

Мы можем настроить это поведение, указав эти задачи в любой из фаз сборки.

Рассмотрим пример, в котором мы привяжем задачу *maven-antrun-plugin:run* к фазам *pre-clean, clean и post-clean*.

Пример:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>ProselyteTutorials</groupId>
    <artifactId>MavenTutorial</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId>
          <version>1.1</version>
          <executions>
            <execution>
              <id>id.pre-clean</id>
```

```
            <phase>pre-clean</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>pre-clean phase</echo>
              </tasks>
            </configuration>
          </execution>
          <execution>
            <id>id.clean</id>
            <phase>clean</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>clean phase</echo>
              </tasks>
            </configuration>
          </execution>
          <execution>
            <id>id.post-clean</id>
            <phase>post-clean</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>post-clean phase</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

После этого выполним в терминале следующую команду:

mvn post-clean

В результате мы получим следующий результат:

[INFO] Scanning for projects...
[INFO]
[INFO] --
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] --
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.pre-clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] pre-clean phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ MavenTutorial ---
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.clean) @ MavenTutorial
[INFO] Executing tasks
[echo] clean phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.post-clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] post-clean phase
[INFO] Executed tasks
[INFO] --
[INFO] BUILD SUCCESS
[INFO] --
[INFO] Total time: 0.417s
[INFO] Finished at: Sun Mar 27 21:38:15 EEST 2021
[INFO] Final Memory: 7M/150M
[INFO] --

Мы также можем выполнить такие же действия для фаз *pre-clean* и *clean*.

Жизненный цикл *Default* (*Build*)

Это основной жизненный цикл *Maven*, который используется для сборки проектов. Он включает в себя 23 фазы (табл. 4.3).

Таблица 4.3

Фазы жизненного цикла *Default* (*Build*)

Фаза жизненного цикла	Описание
validate	Подтверждает, является ли проект корректным, и вся ли необходимая информация доступа для завершения процесса сборки
initialize	Инициализирует состояние сборки, например, различные настройки
generate-sources	Включает любой исходный код в фазу компиляции
process-sources	Обрабатывает исходный код (подготавливает). Например, фильтрует определенные значения
generate-resources	Генерирует ресурсы, которые должны быть включены в пакет
process-resources	Копирует и отправляет ресурсы в указанную директорию. Это фаза перед упаковкой
compile	Комплирует исходный код проекта
process-classes	Обработка файлов, полученных в результате компиляции. Например, оптимизация байт-кода *Java* классов
generate-test-sources	Генерирует любые тестовые ресурсы, которые должны быть включены в фазу компиляции
process-test-sources	Обрабатывает исходный код тестов. Например, фильтрует значения
test-compile	Компилирует исходный код тестов в указанную директорию тестов
process-test-classes	Обрабатывает файлы, полученные в результате компиляции исходного кода тестов
test	Запускает тесты, используя приемлемый фреймворк юнит-тестирования (например, *Junit*)
prepare-package	Выполняет все необходимые операции для подготовки пакета непосредственно перед упаковкой
package	Преобразует скомпилированный код и пакет в дистрибутивный формат. Такие как *JAR*, *WAR* или *EAR*
pre-integration-test	Выполняет необходимые действия перед выполнением интеграционных тестов
integration-test	Обрабатывает и распаковывает пакет, если необходимо, в среду, где будут выполняться интеграционные тесты

Окончание табл. 4.2

Фаза жизненного цикла	Описание
post-integration-test	Выполняет действия, необходимые после выполнения интеграционных тестов. Например, освобождение ресурсов
verify	Выполняет любые проверки для подтверждения того, что пакет пригоден и отвечает критериям качества
install	Устанавливает пакет в локальный репозиторий, который может быть использован как зависимость в других локальных проектах
deploy	Копирует финальный пакет (архив) в удаленный репозиторий, для того чтобы сделать его доступным другим разработчикам и проектам

Необходимо уточнить два момента:

• Когда мы выполняем команду *Maven*, например *install*, то будут выполнены фазы до *install* и фаза *install*.

• Различные задачи *Maven* будут привязаны к различным фазам жизненного цикла Maven в зависимости от типа архива (JAR/WAR/EAR).

В следующем примере мы привязываем задачу *maven-antrun-plugin:run* к нескольким фазам жизненного цикла сборки. Это также позволяет нам вызывать текстовые сообщения, отображая фазу жизненного цикла.

Пример:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>ProselyteTutorials</groupId>
    <artifactId>MavenTutorial</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId>
```

```
<version>1.1</version>
<executions>
  <execution>
    <id>id.validate</id>
    <phase>validate</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>validate phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
    <id>id.compile</id>
    <phase>compile</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>compile phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
    <id>id.test</id>
    <phase>test</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>test phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
```

```
                <id>id.package</id>
                <phase>package</phase>
                <goals>
                  <goal>run</goal>
                </goals>
                <configuration>
                  <tasks>
                    <echo>package phase</echo>
                  </tasks>
                </configuration>
              </execution>
              <execution>
                <id>id.deploy</id>
                <phase>deploy</phase>
                <goals>
                  <goal>run</goal>
                </goals>
                <configuration>
                  <tasks>
                    <echo>deploy phase</echo>
                  </tasks>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
    </build>
</project>
```

После этого выполним следующую команду:

mvn compile

В результате мы получим примерно следующий результат:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------
[INFO]
```

[INFO] --- maven-antrun-plugin:1.1:run (id.pre-clean) @ MavenTutorial ---

[INFO] Executing tasks

[echo] pre-clean phase

[INFO] Executed tasks

[INFO]

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ MavenTutorial ---

[INFO]

[INFO] --- maven-antrun-plugin:1.1:run (id.clean) @ MavenTutorial ---

[INFO] Executing tasks

[echo] clean phase

[INFO] Executed tasks

[INFO]

[INFO] --- maven-antrun-plugin:1.1:run (id.post-clean) @ MavenTutorial ---

[INFO] Executing tasks

[echo] post-clean phase

[INFO] Executed tasks

proselyte@proselyte:~/Programming/Projects/Proselyte/MavenTutorial$ mvn pre-clean

[INFO] Scanning for projects...

[INFO]

[INFO] --

[INFO] Building MavenTutorial 1.0-SNAPSHOT

[INFO] --

[INFO]

[INFO] --- maven-antrun-plugin:1.1:run (id.pre-clean) @ MavenTutorial ---

[INFO] Executing tasks

[echo] pre-clean phase

[INFO] Executed tasks

[INFO] --

[INFO] BUILD SUCCESS

[INFO] --

[INFO] Total time: 0.331s

[INFO] Finished at: Sun Mar 27 21:39:39 EEST 2021

[INFO] Final Memory: 7M/150M
[INFO] --
prose-lyte@proselyte:~/Programming/Projects/Proselyte/MavenTutorial$ mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] --
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] --
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.pre-clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] pre-clean phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ MavenTutorial ---
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] clean phase
[INFO] Executed tasks
[INFO] --
[INFO] BUILD SUCCESS
[INFO] --
[INFO] Total time: 0.405s
[INFO] Finished at: Sun Mar 27 21:39:45 EEST 2021
[INFO] Final Memory: 6M/119M
[INFO] --
prose-lyte@proselyte:~/Programming/Projects/Proselyte/MavenTutorial$ mvn post-clean
[INFO] Scanning for projects...
[INFO]
[INFO] --
[INFO] Building MavenTutorial 1.0-SNAPSHOT

[INFO] --
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.pre-clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] pre-clean phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ MavenTutorial ---
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] clean phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.post-clean) @ MavenTutorial ---
[INFO] Executing tasks
[echo] post-clean phase
[INFO] Executed tasks
proselyte@proselyte:~/Programming/Projects/Proselyte/MavenTutorial$ mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] --
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] --
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.validate) @ MavenTutorial ---
[INFO] Executing tasks
[echo] validate phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-resources-plugin:2.3:resources (default-resources) @ MavenTutorial ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.0.2:compile (default-compile) @ MavenTutorial ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.compile) @ MavenTutorial ---
[INFO] Executing tasks
[echo] compile phase
[INFO] Executed tasks
[INFO] --
[INFO] BUILD SUCCESS
[INFO] --
[INFO] Total time: 0.824s
[INFO] Finished at: Sun Mar 27 22:07:37 EEST 2021
[INFO] Final Memory: 8M/150M
[INFO] --

Жизненный цикл *Site*

Плагин *Maven – Site –* используется для создания докладов, документации, развертывания и т. д.

Он включает в себя такие фазы:

– *pre-site*;
– *site*;
– *post-site*;
– *site-deploy*.

В примере ниже мы прикрепляем задачу *maven-antrun-plugin:run* ко всем фазам жизненного цикла *Site*. Это позволяет нам вызывать текстовые сообщения для отображения фаз жизненного цикла.

Пример:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
<groupId>ProselyteTutorials</groupId>
<artifactId>MavenTutorial</artifactId>
<version>1.0-SNAPSHOT</version>
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-antrun-plugin</artifactId>
         <version>1.1</version>
         <executions>
            <execution>
               <id>id.pre-site</id>
               <phase>pre-site</phase>
               <goals>
                  <goal>run</goal>
               </goals>
               <configuration>
                  <tasks>
                     <echo>pre-site phase</echo>
                  </tasks>
               </configuration>
            </execution>
            <execution>
               <id>id.site</id>
               <phase>site</phase>
               <goals>
                  <goal>run</goal>
               </goals>
               <configuration>
                  <tasks>
                     <echo>site phase</echo>
                  </tasks>
               </configuration>
            </execution>
            <execution>
               <id>id.post-site</id>
               <phase>post-site</phase>
               <goals>
```

```
                <goal>run</goal>
              </goals>
              <configuration>
                <tasks>
                  <echo>post-site phase</echo>
                </tasks>
              </configuration>
            </execution>
            <execution>
              <id>id.site-deploy</id>
              <phase>site-deploy</phase>
              <goals>
                <goal>run</goal>
              </goals>
              <configuration>
                <tasks>
                  <echo>site-deploy phase</echo>
                </tasks>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
</project>
```

Теперь выполним команду *Maven*:

mvn site

В результате мы получим примерно следующий результат:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building MavenTutorial 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.pre-site) @ MavenTuto-
rial ---
[INFO] Executing tasks
```

[echo] pre-site phase
[INFO] Executed tasks
[INFO]
[INFO] --- maven-site-plugin:3.0:site (default-site) @ MavenTutorial ---
[WARNING] Report plugin org.apache.maven.plugins:maven-project-info-reports-plugin has an empty version.
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
[INFO] configuring report plugin org.apache.maven.plugins:maven-project-info-reports-plugin:2.9
[WARNING] No project URL defined - decoration links will not be relativized!
[INFO] Rendering site with org.apache.maven.skins:maven-default-skin:jar:1.0 skin.
[INFO] Generating "Dependency Convergence" report --- maven-project-info-reports-plugin:2.9
[INFO] Generating "Dependency Information" report --- maven-project-info-reports-plugin:2.9
[INFO] Generating "About" report --- maven-project-info-reports-plugin:2.9
[INFO] Generating "Plugin Management" report --- maven-project-info-reports-plugin:2.9
[INFO] Generating "Plugins" report --- maven-project-info-reports-plugin:2.9
[INFO] Generating "Summary" report --- maven-project-info-reports-plugin:2.9
[INFO]
[INFO] --- maven-antrun-plugin:1.1:run (id.site) @ MavenTutorial ---
[INFO] Executing tasks
[echo] site phase
[INFO] Executed tasks
[INFO] --
[INFO] BUILD SUCCESS

[INFO] --
[INFO] Total time: 2.717s
[INFO] Finished at: Sun Mar 27 23:02:41 EEST 2021
[INFO] Final Memory: 17M/268M
[INFO] --

ГЛАВА 5. ИНСТРУМЕНТАРИЙ УПРАВЛЕНИЯ ПРОЦЕССОМ РАЗВЕРТЫВАНИЯ

5.1. Знакомство с *Jenkins*

Jenkins – проект для непрерывной интеграции с открытым исходным кодом, написанный на *Java* [11].

Почему стоит использовать именно *Jenkins*:

– бесплатный и надежный;

– большое количество мануалов по работе, а значит, учиться работать с ним проще;

– легко забэкапить и развернуть на другой машине в течение 5 мин;

– осуществлять управление можно конфигурированием и перемещением *xml* файлов.

Из минусов отметим:

– частый выход новых версий, если используется не *LTS*;

– интеграция с различными сервисами завязана на множество различных плагинов;

– иногда может появится задача, которая требует нестандартного подхода.

Идеальный процесс разработки можно представить так:

– разработчик отправляет код в репозиторий;

– на сервере непрерывной интеграции изменения объединяются с основным кодом, выполняются юнит-тесты;

– полученные на предыдущем шаге артефакты загружаются в отдельную тестовую среду, где приложение тестируется автотестами;

– дальше все проверяется для попадания на продакшн;

– развертывание на продакшене.

Установка *Jenkins*

1. Необходимо установить *java* и инструмент архивации, в нашем случае – *unzip*.

2. Разархивировать ранее сохраненный файл (предварительно перейдя в каталог с сохранным архивом):

unzip Jenkins.zip -d %path_To_Jenkins%

3. Перейти в каталог *Jenkins* и выполнить запуск *Jenkins* командой:

cd %path_To_Jenkins% && java -jar jenkins.war

4. Зайти в *WEB* интерфейс по порту 8080.

Если вы работаете с *linux* машиной по *ssh*, то после закрытия соединения, вероятно, *Jenkins* будет остановлен. Для того чтобы избежать такого поведения, можно использовать следующий прием: выполнить команду с добавлением символа & – в результате команда будет выполняться в фоне, а управление будет возвращено обратно в командную строку. Для того чтобы при отключении от удаленной системы не завершались запущенные задачи, можно использовать утилиту *nohup*, которая позволяет процессам продолжать работу даже после того, как вы выйдете из системы:

nohup java -jar jenkins.war &

Добавление *Jenkins* в службы

Для того чтобы в *linux* системах избежать ситуации, описанной в предыдущем абзаце, можно установить *Jenkins* как службу, для того чтобы каждый раз после перезагрузки *Jenkins* стартовал автоматически. Для этого необходимо создать файл */etc/systemd/system/jenkins.service* командой:

sudo cat /etc/systemd/system/jenkins.service
sudo vi /etc/systemd/system/jenkins.service

и добавить содержимое в *jenkins.service*:

[Unit]
Description=Jenkins Daemon
[Service]
Environment="JENKINS_HOME=%path_To_Jenkins%"
ExecStart=/usr/bin/java -jar %path_To_Jenkins%/jenkins.war
*User=*имя текущего пользователя
[Install]
WantedBy=multi-user.target

Выполнить перезапуск службы сервисов:

sudo systemctl daemon-reload

Команда для запуска сервиса *jenkins*:

sudo systemctl start jenkins.service

Команда для перезапуска:

sudo systemctl restart jenkins.service

Важно отметить, что файл *jenkins.war* может быть расположен где угодно. Для того чтобы «подхватились» текущие параметры проекта, можно использовать команду, которая выполнит монтирование раздела *Jenkins* к работающему *jenkins*:

sudo mount -o bind /%path_To_Jenkins%/ ~/.jenkins/

Такой вариант будет работать только до перезагрузки системы, поэтому можно создать симлинк в директории ~/:

cd ~/ && sudo ln -s /%path_To_Jenkins%/ .jenkins

Добавление и подготовка к работе *node* в *Jenkins*

Для добавления *slave node* в *Jenkins* можно добавить каждую машину руками, но из-за этого требуется постоянно заходить на машину и выполнять одни и те же операции, а это приводит к мысли, что все можно максимально автоматизировать. Для этого достаточно создать несколько *Job*, которые будут последовательно выполнять:

– заказ машин для добавления в качестве *slave node* в *Jenkins*;

– добавлять ранее заказанные машины в *Jenkins* с использованием *rest api* или *Jenkins cli*;

– выполнять развертывание необходимого окружения.

Все выше названные действия можно выполнять, используя дополнительные инструменты: *ansible* – для развертывания и настройки необходимых параметров и *docker* – как для развертывания *Jenkins*, так и для установки окружения на *slave nodes*.

5.2. Работа с плагинами в *Jenkins*

На сегодняшний день для *Jenkins* написано более 1500 плагинов, позволяющих решить практически любую проблему развертывания, разработки или тестирования. Они выполняют множество функций, включая непрерывную доставку, передачу файлов из сборки в сборку, отображение проектов в *JIRA* и многое другое.

Плагины можно устанавливать, обновлять и удалять с помощью экрана «Управление плагинами» (рис. 5.1).

Здесь можно установить самые разнообразные сторонние плагины прямо от различных инструментов управления исходным кодом, таких как *Git*, *Mercurial* или *ClearCase*, до отчетов о качестве кода и показателях покрытия кода.

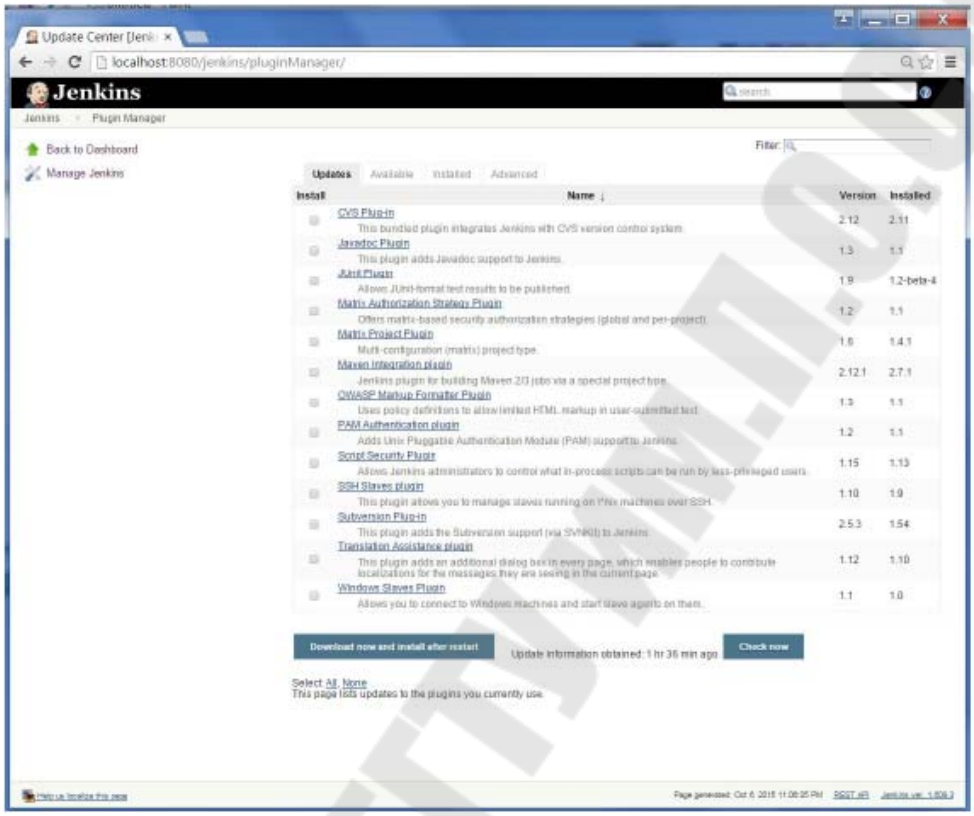

Рис. 5.1. Управление плагинами

Опишем наиболее популярные плагины:

• *Pipeline*. Позволяет создавать сценарий сборки, состоящий из одной или нескольких задач, и визуально отслеживать рабочий процесс. Можно увидеть, какие задания уже запущены или все еще находятся в очереди на исполнение. Самое лучшее в *pipeline* – это интеграция с системой контроля версий, благодаря чему вы можете следить за изменениями кода, выполнять проверку и многое другое. Конвейер сборки может стать важной частью вашей рабочей модели.

• *JIRA*. Для эффективной совместной работы жизненно необходимо отслеживать различные проекты и организовывать их в соответствии с выбранной рабочей моделью. Многие команды используют *JIRA* в качестве центрального инструмента управления одиночной и совместной деятельностью. Чтобы последовать их примеру, вам достаточно просто подключить его к вашей *Jenkins*-конфигурации.

Используя механизм *fingerprinting*, можно также записать в *JIRA* сборки других проектов, связанных с текущим. С помощью этого плагина вы сможете поддерживать чистый и организованный процесс работы над длительным проектом с минимальными ошибками.

• *Build Failure Analyzer*. Находит причины ошибочных сборок с помощью базы знаний, которая изначально пуста. После того как вы ее наполните из логов, плагин сможет определять, чем вызван конкретный сбой. Кроме того, можно перейти к нужной строке журнала одним щелчком мыши.

• *WhiteSource Jenkins*. Упрощает управление компонентами с открытым исходным кодом. Этот плагин позволяет отслеживать лицензии, риски и требования. Он автоматически обновляет репозиторий *WhiteSource* всякий раз, когда добавляется новый *open-source* компонент, что гарантирует быструю и актуальную обработку. Вы получаете уведомление, когда для любого из них доступен новый релиз, а также своевременно узнаете об исправлениях уязвимостей в системе безопасности, проблемах с производительностью и других ошибках.

• *Copy Artifact Plugin*. Позволяет в процессе сборки переносить артефакты между проектами. Например, вы можете скопировать необходимые файлы в другое задание и запустить тесты. Основная концепция плагина заключается в том, что артефакты могут повторно использоваться в разных сборках. Создавая архив, вы поручаете *Jenkins* сохранить его, и после всех необходимых настроек можете начать использовать.

• *GitHub Pull Request Builder*. Автоматизирует просмотр кода из *pull*-запросов на *Github*. Этот плагин объединяет и запускает сборку, а также собирает необходимый статический анализ и предоставляет статус запроса. Можно даже автоматически сливать новый код, если сборка утверждена.

5.3. Работа с трубопроводом (*pipeline*)

Jenkins Pipeline – набор плагинов, позволяющий определить жизненный цикл сборки и доставки приложения как код. Он представляет собой *Groovy*-скрипт с использованием *Jenkins Pipeline DSL* и хранится стандартно в системе контроля версий.

Существует два способа описания пайплайнов – скриптовый и декларативный:

1. *Scripted*:

```
node {
  stage('Example') {
     try {
        sh 'exit 1'
     }
     catch (exc) {
        throw exc
     }
  }
}
```

2. *Declarative*

```
pipeline {
  agent any
  stages {
     stage("Stage name") {
        steps {}
     }
  }
}
```

Они оба имеют структуру, но в скриптовом она вольная — достаточно указать, на каком слейве запускаться (*node*) и стадию сборки (*stage*), а также написать *Groovy*-код для запуска атомарных степов.

Декларативный пайплайн определен более жестко и, соответственно, его структура читается лучше.

Рассмотрим подробнее декларативный пайплайн.

1. В структуре должна быть определена директива ***pipeline***.

2. Также нужно определить, на каком агенте (***agent***) будет запущена сборка.

3. Дальше идет определение ***stages***, которые будут содержаться в пайплайне, и обязательно должен быть конкретный стейдж с названием ***stage(«name»)***. Если имени нет, тест упадет в *runtime* с ошибкой «Добавьте имя стейджа».

4. Обязательно должна быть директива ***steps***, в которой уже содержатся атомарные шаги сборки. Например, вы можете вывести в консоль «*Hello*»:

```
pipeline { // определение декларативного pipeline
  agent any // определяет, на каком агенте будет запущена сборка
```

```
  stages { // содержит стейджи сборки
     stage("Stage name") { // отдельный стейдж сборки
        steps { // набор шагов в рамках стейджа
           echo "Hello work" // один из шагов сборки
        }
     }
  }
}
```

Мне нравится декларативный вид пайплайна тем, что он позволяет определить действия после каждого стейджа или, например, после всей сборки. Я рекомендую использовать его при описании пайплайнов на верхнем уровне:

```
pipeline {
  stages {
     stage("Post stage") {
        post { // определяет действия по завершении стейджа
           success { // триггером исполнения секции является состояние сборки
              archiveArtifacts artifacts: '**/target/*'
           }
        }
     }
  }
  post { // после всей сборки
     cleanup {
        cleanWs()
     }
  }
}
```

Если сборка и стейдж завершились успешно, можно сохранить артефакты или почистить *workspace* после сборки. Если же при таких условиях использовался бы скриптовый пайплайн, пришлось бы за этим «флоу» следить самостоятельно, добавляя обработку исключений, условия и т. д.

ГЛАВА 6. УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ

6.1. Управление конфигурацией

Управление конфигурацией – это процесс учета изменений, вносимых в систему с целью сохранения ее целостности. Обычно используются инструменты и методы, способствующие автоматизации процесса и наблюдению состояния системы. Хотя эта концепция родилась не в ИТ-индустрии, термин стал широко использоваться для обозначения управления конфигурацией сервера [12].

В контексте серверов управление конфигурацией также часто называют ИТ-автоматизацией или оркестрацией сервера. Оба термина освещают практические аспекты управления конфигурацией, а также возможность контроля нескольких систем с центрального сервера.

Вы узнаете о преимуществах использования инструментов управления конфигурацией для автоматизации настроек серверной инфраструктуры на примере системы управления *Ansible*.

Преимущества использования инструментов управления конфигурацией

На рынке присутствует целый ряд инструментов управления конфигурацией разного уровня сложности и с разными архитектурами. Хотя все эти инструменты отличаются характеристиками и принципами работы, все они выполняют одну и ту же функцию: обеспечивают соответствие системы параметрам, заявленным в наборе скриптов конфигурирования.

Преимущества управления конфигурацией серверов состоят в способности определять вашу инфраструктуру как код. Это позволяет:

– использовать систему контроля версий для отслеживания любых изменений инфраструктуры;

– повторно использовать скрипты конфигурирования для нескольких серверных сред, например, для разработки, тестирования и производства;

– предоставлять сотрудникам общий доступ к скриптам конфигурирования для упрощения сотрудничества в стандартизированной среде разработки;

– упрощать процесс дублирования серверов для ускорения восстановления в случае сбоя системы.

К тому же инструменты управления конфигурацией предлагают возможность централизованного контроля неограниченного числа серверов из одной точки. Это может значительно повысить производительность и укрепить целостность инфраструктуры.

6.2. Знакомство с Ansible

Ansible – это современный инструмент управления конфигурацией, который упрощает настройку и удаленное администрирование серверов. Минималистичный дизайн инструмента обеспечивает простое и понятное использование.

Пользователи пишут скрипты конфигурирования *Ansible* в удобном формате сериализации данных *YAML*, который не привязывается к какому-либо языку программирования. Это позволяет пользователям интуитивно создавать сложные скрипты конфигурирования, в отличие от аналогичных инструментов такой же категории [13].

Ansible не требует установки специального программного обеспечения на узлах, где будет работать эта система. Контрольный механизм, настроенный в программном обеспечении *Ansible*, связывается с узлами через стандартные каналы *SSH*.

Как инструмент управления конфигурацией и система автоматизации *Ansible* имеет все функции, присутствующие в других инструментах этой же категории, но при этом данная система ориентируется на простоту использования и производительность.

Идемпотентное поведение

Ansible отслеживает состояние ресурсов управляемых систем для недопущения повторения задач, которые выполнялись ранее. Если пакет уже установлен, система не будет пытаться установить его снова. Основной задачей является то, что при каждом исполнении система достигает (или сохраняет) желаемое состояние, даже если вы запускаете ее несколько раз. Это означает, что системе *Ansible* и другим инструментам управления конфигурацией присуще идемпотентное поведение. При запуске плейбука вы увидите статус каждой выполняемой задачи и указание, приводит ли выполненная задача к изменению системы.

Поддержка переменных, условий и циклов

При написании скриптов автоматизации *Ansible* вы можете использовать переменные, условия и циклы, чтобы сделать процесс автоматизации более универсальным и эффективным.

Системные сведения

Ansible собирает серию подробных данных об управляемых узлах, например, сетевых интерфейсах и операционной системе, и обозначает эти данные как глобальные переменные, называемые системными сведениями. Сведения можно использовать внутри плейбуков, чтобы обеспечить универсальность и адаптивность автоматизации, работающей по-разному в зависимости от системы конфигурирования.

Система шаблонов

Ansible использует систему шаблонов *Jinja*2 *Python*, разрешающую динамические выражения и доступ к переменным. Шаблоны можно использовать для облегчения настройки файлов и служб конфигурации. Например, вы можете использовать шаблон для настройки нового виртуального хоста в *Apache*, а также использовать такой же шаблон для нескольких установок сервера.

Поддержка расширений и модулей

Ansible поставляется с сотнями встроенных модулей, упрощающих автоматизацию стандартных задач администрирования, таких как установка пакетов с помощью *apt* и синхронизация файлов через *rsync*, а также работа с популярными программами, например, системами базы данных (*MySQL*, *PostgreSQL*, *MongoDB* и др.) и инструментами управления зависимостями (*PHP composer*, *Ruby gem*, *Nodenpm* и др.).

Помимо этого существует ряд способов расширения системы *Ansible*. Это плагины и модули, которые необходимы для обеспечения пользовательских функций, не включенных по умолчанию.

Вы также можете использовать модули и плагины сторонних организаций, представленные на портале *Ansible Galaxy*.

Знакомство с концепциями *Ansible*

Давайте рассмотрим терминологию и концепции *Ansible*, чтобы познакомиться с терминами, которые будут употребляться в этой серии материалов.

Узел управления

Узел управления – это устройство с установленной и настроенной для подключения к вашему серверу системой *Ansible*. Вы можете использовать несколько узлов управления, а любая система, на которой возможен запуск *Ansible*, может быть настроена как узел управления, включая персональные компьютеры или ноутбуки на базе операционных систем *Linux* или *Unix*. На данный момент *Ansible* нельзя

установить на хосте *Windows*, но вы можете обойти это ограничение, настроив виртуальную машину под управлением *Linux* и запустив на ней *Ansible*.

Управляемые узлы

Системы, которыми вы управляете с помощью *Ansible*, называются управляемыми узлами. Для работы *Ansible* необходимо, чтобы управляемые узлы были доступны через каналы *SSH* и на них был установлен *Python* 2 (версия 2.6 или выше) или *Python* 3 (версия 3.5 или выше).

Ansible поддерживает различные операционные системы в качестве управляемых узлов, включая серверы *Windows*.

Файлы инвентаризации

Файл инвентаризации содержит список хостов, которыми вы будете управлять при помощи *Ansible*. Хотя *Ansible* обычно создает файл инвентаризации по умолчанию при установке, вы можете использовать файлы инвентаризации для каждого проекта отдельно. Это обеспечит лучшее разделение инфраструктуры и позволит избежать ошибочного выполнения команд или плейбуков не на том сервере. Статические файлы инвентаризации обычно создаются как файлы *.ini*, но вы также можете использовать динамически сгенерированные файлы, написанные на любом языке программирования, способном возвращать *JSON*.

Задачи

В *Ansible* задача – это отдельная часть работы, которую нужно выполнить на управляемом узле. Каждое действие, которое нужно выполнить, определяется как задача. Задачи можно выполнять как единичные действия через ситуативные команды или же включать их в плейбуки в качестве части скрипта автоматизации.

Плейбук

Плейбук состоит из списка заданных задач и нескольких директив, указывающих, какой хост является целью автоматизации, а также нужно ли использовать систему эскалации привилегий для выполнения этих задач. В плейбук также могут быть включены дополнительные разделы для определения переменных или включения файлов. *Ansible* выполняет задачи последовательно, а полноценное выполнение плейбука называется «плей». Плейбуки записываются в формате *YAML*.

Обработчики

Обработчики используются для выполнения действий со службами, например, при перезагрузке или остановке службы, которая ак-

тивно работает на системе управляемого узла. Обработчики обычно запускаются с помощью задач, а их использование происходит в конце плея, после выполнения всех задач. В этом случае, если перезагрузку службы начинает более чем одна задача, служба запустится только один раз после выполнения всех команд. Хотя поведение обработчика по умолчанию более эффективно, также возможно принудительное немедленное выполнение обработчика, если этого требует задача.

Роли

Роль – это набор плейбуков и связанных файлов, организованных в предопределенную структуру, известную *Ansible*. Роли упрощают повторное использование плейбуков и превращают их в пакеты детализированной автоматизации с общим доступом для конкретных целей, таких как установка веб-сервера, среды *PHP* или настройка сервера *MySQL*.

Ansible – это минималистичный и простой для изучения инструмент ИТ-автоматизации благодаря использованию *YAML* в скриптах конфигурирования. Он имеет большое количество встроенных модулей, которые можно использовать для абстрактных задач, таких как установка пакетов и работа с шаблонами. Его упрощенные требования к инфраструктуре и доступный синтаксис подходят тем, кто только начинает работать с управлением конфигурацией.

Установка *Ansible*

Из исходников. Ветка *devel* всегда стабильна, так что используем ее. Возможно, вам нужно будет установить *git* (*sudo apt-get install git* на *Debian*/*Ubuntu*):

```
git clone git://github.com/ansible/ansible.git
cd ./ansible
```

Теперь можно загрузить окружение *Ansible*:

```
source ./hacking/env-setup
```

Из *deb* пакета:

```
sudo apt-get install make fakeroot cdbs python-support
git clone git://github.com/ansible/ansible.git
cd ./ansible
make deb
sudo dpkg -i ../ansible_1.1_all.deb (version may vary)
```

В этом пособии предполагается, что вы использовали именно этот способ.

Установка *Vagrant*

Vagrant позволяет с легкостью создавать виртуальные машины и запускать их на *VirtualBox*. *Vagrantfile* идет в комплекте с пособием.

Чтобы запустить *Vagrant*, вам нужно установить:

– *VirtualBox*;

– *Ruby* (скорее всего уже установлено на вашей системе);

– *Vagrant* 1.1+.

Теперь инициализируйте виртуальную машину с помощью следующей команды:

vagrant up

Добавление *SSH*-ключей на виртуальной машине

Чтобы продолжить, вам нужно добавить свои ключи в *authorized_keys root*'а на виртуальной машине. Это не обязательно (*Ansible* может использовать *sudo* и авторизацию по паролю), но так будет намного проще.

Ansible идеально подходит для этой задачи, поэтому используем его:

ansible-playbook -c paramiko -i step-00/hosts step-00/setup.yml --ask-pass --sudo

В качестве пароля введите *vagrant*. Если возникнут ошибки «*Connections refused*», то проверьте настройки фаервола.

Теперь добавьте свои ключи в *ssh-agent* (*ssh-add*).

Inventory

Теперь нам нужно подготовить файл *inventory*. Место по умолчанию – это */etc/ansible/hosts*. Но вы можете настроить *Ansible* так, чтобы использовался другой путь. Для этого используется переменная окружения (*ANSIBLE_HOSTS*) или флаг *-i*.

Мы создали такой файл *inventory*:

host0.example.org ansible_ssh_host=192.168.33.10
ansible_ssh_user=root
host1.example.org ansible_ssh_host=192.168.33.11
ansible_ssh_user=root
host2.example.org ansible_ssh_host=192.168.33.12
ansible_ssh_user=root

ansible_ssh_host – это специальная *переменная*, которая содержит *IP*-адрес узла, к которому будет создаваться соединение. В данном случае она не обязательна, если вы используете *gem vagrant-hostmaster*.

Также вам нужно будет менять *IP*-адреса, если вы устанавливали и настраивали свою виртуальную машину с другими адресами.

ansible_ssh_user – это еще одна специальная *переменная*, которая говорит *Ansible*'у подключаться под указанным аккаунтом (юзером). По умолчанию *Ansible* использует ваш текущий аккаунт или другое значение по умолчанию, указанное в ~/*.ansible.cfg* (*remote_user*).

Проверка

Теперь, когда *Ansible* установлен, давайте проверим, что все работает:

ansible -m ping all -i step-01/*hosts*

Здесь *Ansible* попытается запустить модуль *ping* (подробнее о модулях позже) на каждом хосте. Вывод должен быть примерно таким:

```
host0.example.org | success >> {
    "changed": false,
    "ping": "pong"
}

host1.example.org | success >> {
    "changed": false,
    "ping": "pong"
}

host2.example.org | success >> {
    "changed": false,
    "ping": "pong"
}
```

Все три хоста работают и *Ansible* может общаться с ними.

Общение с узлами

В прошлой команде *-m ping* означал «используй модуль *ping*». Это один из множества модулей, доступных в *Ansible*. Модуль *ping* очень прост, он не требует никаких аргументов. Модули, требующие аргументы, могут получить их через *-a*. Давайте взглянем на несколько модулей.

Модуль *shell*

Этот модуль позволяет запускать *shell*-команды на удаленном узле:

ansible -i step-02/*hosts -m shell -a 'uname -a' host*0.*example.org*

Вывод должен быть вроде:

*host*0.*example.org* | *success* | *rc*=0 >>
*Linux host*0.*example.org* 3.2.0-23-*generic-pae* #36-*Ubuntu SMP Tue Apr* 10 22:19:09 *UTC* 2012 *i*686 *i*686 *i*386 *GNU*/*Linux*

Модуль *copy*

Модуль *copy* позволяет копировать файл из управляющей машины на удаленный узел. Представим, что нам нужно скопировать наш /*etc*/*motd* в /*tmp* узла:

ansible -i step-02/*hosts -m copy -a* '*src*=/*etc*/*motd dest*=/*tmp*/' *host*0.*example.org*

Вывод:

```
host0.example.org | success >> {
    "changed": true,
    "dest": "/tmp/motd",
    "group": "root",
    "md5sum": "d41d8cd98f00b204e9800998ecf8427e",
    "mode": "0644",
    "owner": "root",
    "size": 0,
    "src":   "/root/.ansible/tmp/ansible-1362910475.9-246937081757218/
    motd",
    "state": "file"
}
```

Ansible (точнее, модуль *copy*, запущенный на узле) ответил «кучей» полезной информации в формате *JSON*. Позже мы увидим, как это можно использовать.

У *Ansible* есть огромный список модулей, который покрывает практически все, что можно делать в системе. Если вы не нашли подходящего модуля, то написание своего модуля – довольно простая задача (и не обязательно писать его на *Python*, главное, чтобы он понимал *JSON*).

Много хостов, одна команда

Допустим, мы хотим собрать факты про узел и, например, хотим узнать, какая версия *Ubuntu* установлена на узлах. Это довольно легко:

ansible -i step-02/*hosts -m shell -a 'grep DISTRIB_RELEASE /etc/lsb-release' all*

all означает «все хосты в файле *inventory*». Вывод будет примерно таким:

*host*1.*example.org* | *success* | *rc*=0 >>
DISTRIB_RELEASE=12.04
*host*2.*example.org* | *success* | *rc*=0 >>
DISTRIB_RELEASE=12.04
*host*0.*example.org* | *success* | *rc*=0 >>
DISTRIB_RELEASE=12.04

Больше фактов

Однако, если нам нужно больше информации (*IP*-адреса, размеры ОЗУ и пр.), такой подход может быстро оказаться неудобным. Решение – использовать модуль *setup*. Он специализируется на сборе *фактов* с узлов.

Попробуйте:

ansible -i step-02/*hosts -m setup host*0.*example.org*

Ответ:

```
"ansible_facts": {
   "ansible_all_ipv4_addresses": [
      "192.168.0.60"
   ],
   "ansible_all_ipv6_addresses": [],
   "ansible_architecture": "x86_64",
   "ansible_bios_date": "01/01/2007",
   "ansible_bios_version": "Bochs"
   },
   ---snip---
   "ansible_virtualization_role": "guest",
   "ansible_virtualization_type": "kvm"
},
"changed": false,
"verbose_override": true
```

Вывод был сокращен для простоты, но вы можете узнать много интересного из этой информации. Вы также можете фильтровать ключи, если вас интересует что-то конкретное.

Например, вам нужно узнать, сколько памяти доступно на всех хостах. Это легко: запустите *ansible -i step*-02/*hosts -m setup -a 'filter=ansible_memtotal_mb' all*:

```
host2.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}
host1.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}
host0.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}
```

Заметьте, что узлы ответили не в том порядке, в котором они отвечали выше. *Ansible* общается с хостами параллельно.

Кстати, при использовании модуля *setup* можно указывать * в выражении *filter=*. Как в *shell*.

Выбор хостов

Мы видели, что *all* означает «все хосты», но в *Ansible* есть много иных способов выбирать хосты:

host0.example.org:host1.example.org будет запущен на host0.example.org и нahost1.example.org

host.example.org будет запущен на всех хостах, названия которых начинается с 'host' и заканчивается на '.example.org' (тоже как в shell)*

Группировка хостов

Хосты в *inventory* можно группировать. Например, можно создать группу *debian*, группу *web-servers*, группу *production* и т. д.:

[debian]
*host*0*.example.org*
*host*1*.example.org*
*host*2*.example.org*
Можно даже сократить:
[debian]
*host[*0-2*].example.org*

Если хотите задавать дочерние группы, используйте *[groupname*:*children]* и добавьте дочерние группы в него. Например, у нас есть разные дистрибутивы Линукса, их можно организовать следующим образом:

[ubuntu]
*host*0*.example.org*
[debian]
*host[*1-2*].example.org*
[linux:*children]*
ubuntu
debian

Установка переменных

Вы можете добавлять переменные для хостов в нескольких местах: в файле *inventory*, файлах переменных хостов, файлах переменных групп и др.

Обычно я задаю все переменные в файлах переменных групп/хостов (подробнее об этом позже). Однако зачастую я использую переменные напрямую в файле *inventory*, например, *ansible_ssh_host*, которая задает *IP*-адрес хоста. По умолчанию *Ansible* резолвит имена хостов при соединении по *SSH*. Но когда вы инициализируете хост, он, возможно, еще не имеет *IP*-адреса. *ansible_ssh_host* будет полезен в таком случае.

При использовании команды *ansible-playbook* (а не обычной команды *ansible*) переменные можно задавать с помощью флага *--extra-vars* (или *-e*). О команде *ansible-playbook* мы поговорим в следующем шаге.

ansible_ssh_port, используется, чтобы задать порт соединения по *SSH*:

[ubuntu]

*host*0.*example.org* *ansible_ssh_host*=192.168.0.12 *ansible_ssh_port*=2222

Ansible ищет дополнительные переменные в файлах переменных групп и хостов. Он будет искать эти файлы в директориях *group_vars* и *host_vars*, внутри директории, где расположен главный файл *inventory*.

Ansible будет искать файлы по имени. Например, при использовании упомянутого ранее файле *inventory*, *Ansible* будет искать переменные *host*0.*example.org* в файлах:

– *group_vars/linux*;

– *group_vars/ubuntu*;

– *host_vars/host0.example.org*.

Если этих файлов не существует, ничего не произойдет, но если они существуют, они будут использованы.

Теперь, когда мы познакомились с модулями, инвентаризацией и переменными, давайте, наконец, узнаем о настоящей «мощи» *Ansible* с плейбуками.

6.3. Сценарии в Ansible (*playbook*)

Концепция плейбуков очень проста: это просто набор команд *Ansible* (задач, *tasks*), похожих на те, что мы выполняли с утилитой *ansible*. Эти задачи направлены на конкретные наборы узлов/групп.

Пример с Apache. Продолжаем с допущением, что ваш файл *inventory* выглядит так (назовем его *hosts*):

[web]

*host*1.*example.org*

и все хосты – это системы на основе *Debian*.

Примечание. Помните, что вы можете (и в нашем упражнении мы делаем это) использовать *ansible_ssh_host*, чтобы задать реальный *IP*-адрес хоста. Вы также можете изменять *inventory* и использовать реальный *hostname*. В любом случае используйте машину, с которой безопасно экспериментировать. На реальных хостах мы также добавляем *ansible_ssh_user=root*, чтобы избежать потенциальных проблем с разными конфигурациями по умолчанию.

Давайте соберем плейбук, который установит *Apache* на машины группы *web*:

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true
```

Нам всего лишь нужно сказать, что мы хотим сделать, используя правильные модули *Ansible*. Здесь мы используем модуль *apt*, который может устанавливать пакеты *Debian*. Мы также просим этот модуль обновить кэш.

Нам нужно имя для этой задачи. Это не обязательно, но желательно для вашего же удобства.

Теперь можно запустить плейбук (назовем его *apache.yml*):

ansible-playbook -i step-04/*hosts -l host*1.*example.org step*-04/*apache.yml*

Здесь *step*-04/*hosts* – это файл *inventory*, *-l* ограничивает запуск хостом *host*1.*example.org*, а *apache.yml* – это наш плейбук.

При запуске команды будет вывод подобный этому:

```
PLAY [web] *********************
GATHERING FACTS *********************
ok: [host1.example.org]
TASK: [Installs apache web server] *********************
changed: [host1.example.org]
PLAY RECAP *********************
host1.example.org  : ok=2  changed=1  unreachable=0  failed=0
```

Примечание. Возможно, вы заметите проходящую мимо «корову», если у вас установлен *cowsay* :-). Если она вам не нравится, можно отключить ее так: *export ANSIBLE_NOCOWS*="1".

Давайте проанализируем вывод строчку за строчкой.

*PLAY [web] *********************

Ansible говорит нам, что *play* выполняется в группе *web*. *Play* – это набор инструкций *Ansible*, связанных с хостом. Если бы у нас был другой -*host*: *blah* в плейбуке, он бы тоже вывелся (но после того, как первый *play* завершен):

```
GATHERING FACTS *********************
ok: [host1.example.org]
```

Перед каждым воспроизведением *Ansible* запускает его на каждом хосте и собирает факты. Если это не требуется (скажем, потому что вам не нужна никакая информация о хосте), можно добавить *gather_facts*: *no* под строкой хоста (на том же уровне, где находится *tasks*:).

TASK: *[Installs apache web server]* **********
changed: [host1.example.org]

Теперь самое главное: наша первая и единственная задача запущена, и так как там указано *changed*, мы знаем, что она изменила что-то на хосте *host*1.*example.org*.

PLAY RECAP **********
host1.example.org : ok=2 changed=1 unreachable=0 failed=0

Наконец, *Ansible* выводит выжимку того, что произошло: две задачи были выполнены, и одна из них изменила что-то на хосте (это была наша задача *apache*; модуль *setup* ничего не меняет).

Давайте запустим это еще раз и посмотрим, что произойдет:

$ *ansible-playbook -i step*-04/*hosts -l host*1.*example.org step*-04/*apache.yml*

PLAY [web] **********
GATHERING FACTS **********
ok: *[host*1.*example.org]*
TASK: *[Installs apache web server]* **********
ok: [host1.example.org]
PLAY RECAP **********
*host*1.*example.org* : *ok*=2 *changed*=0 *unreachable*=0 *failed*=0

Теперь *changed* равен '0'. Это совершенно нормально и является одной из главных особенностей *Ansible*: плейбук будет делать что-то, только если есть что делать. Это называется *идемпотентностью*. Это значит, что можно запускать плейбук сколько угодно раз, но в итоге мы будем иметь машину в одном и том же состоянии.

Улучшаем набор apache. Мы установили *apache*, давайте теперь настроим *virtualhost*.

Нам нужен один виртуальный хост на сервере, но мы хотим сменить дефолтный на что-то более конкретное. Поэтому нам придется удалить текущий *virtualhost*, отправить наш *virtualhost*, активировать его и перезапустить *apache*.

Давайте создадим директиорию под названием *files* и добавим нашу конфигурацию для *host*1.*example.org*, назовем ее *awesome-app*:

```
<VirtualHost *:80>
  DocumentRoot /var/www/awesome-app
  Options -Indexes
  ErrorLog /var/log/apache2/error.log
  TransferLog /var/log/apache2/access.log
</VirtualHost>
```

Теперь небольшое обнуление плейбука и все готово:

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true
    - name: Push default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/
mode=0640
    - name: Deactivates the default virtualhost
      command: a2dissite default
    - name: Deactivates the default ssl virtualhost
      command: a2dissite default-ssl
    - name: Activates our virtualhost
      command: a2ensite awesome-app
      notify:
        - restart apache
  handlers:
    - name: restart apache
      service: name=apache2 state=restarted
```

***Запускаем*:**

```
$ ansible-playbook -i step-05/hosts -l host1.example.org step-
05/apache.yml
PLAY [web] ********************
GATHERING FACTS ********************
ok: [host1.example.org]
TASK: [Installs apache web server] ********************
ok: [host1.example.org]
```

TASK: [Push default virtual host configuration] **********

changed: *[host*1*.example.org]*

TASK: *[Deactivates the default virtualhost]* **********

changed: [host1.example.org]

TASK: [Deactivates the default ssl virtualhost] **********

changed: *[host*1*.example.org]*

TASK: *[Activates our virtualhost]* **********

changed: [host1.example.org]

NOTIFIED: [restart apache] **********

changed: *[host*1*.example.org]*

PLAY RECAP **********

host1.example.org : ok=7 changed=5 unreachable=0 failed=0

Но, если задуматься, мы немного опережаем события. Не нужно ли проверить корректность конфигурации перед тем, как перезапускать *apache*, чтобы не нарушать работоспособность сервиса в случае, если конфигурация содержит ошибку.

Перезапуск в случае ошибки конфигурации. Мы установили *apache*, изменили *virtualhost* и перезапустили сервер. Но что, если мы хотим перезапускать сервер, только когда конфигурация корректна?

Откатываемся, если есть проблемы.

Ansible содержит «классную» особенность: он остановит всю обработку, если что-то пошло не так. Мы используем эту особенность, чтобы остановить плейбук, когда конфигурация невалидна.

Давайте изменим файл конфигурации виртуального хоста *awesome-app* и сломаем его:

```
<VirtualHost *:80>
  RocumentDoot /var/www/awesome-app
  Options -Indexes
  ErrorLog /var/log/apache2/error.log
  TransferLog /var/log/apache2/access.log
</VirtualHost>
```

Как я сказал, если задача не может исполниться, обработка останавливается. Так что нужно удостовериться в валидности конфигурации перед перезапуском сервера. Мы также начнем с добавления

виртуального хоста до удаления дефолтного виртуального хоста, так что последующий перезапуск (возможно, сделанный напрямую на сервере) не сломает *apache*.

Нужно было сделать это в самом начале. Так как мы уже запускали этот плейбук, дефолтный виртуальный хост уже деактивирован. Это не проблема: этот плейбук можно использовать на других хостах, так что давайте защитим их:

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true
    - name: Push future default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/
mode=0640
    - name: Activates our virtualhost
      command: a2ensite awesome-app
    - name: Check that our config is valid
      command: apache2ctl configtest
    - name: Deactivates the default virtualhost
      command: a2dissite default
    - name: Deactivates the default ssl virtualhost
      command: a2dissite default-ssl
    notify:
      - restart apache
  handlers:
    - name: restart apache
      service: name=apache2 state=restarted
```

***Запускаем*:**

```
$ ansible-playbook -i step-06/hosts -l host1.example.org step-
06/apache.yml
PLAY [web] ********************
GATHERING FACTS ********************
ok: [host1.example.org]
TASK: [Installs apache web server] ********************
ok: [host1.example.org]
TASK: [Push future default virtual host configuration]
********************
changed: [host1.example.org]
```

TASK: *[Activates our virtualhost]* **********
changed: [host1.example.org]
TASK: [Check that our config is valid] **********
failed: *[host*1*.example.org]* => {"*changed*": *true*, "*cmd*": *["apache*2*ctl*", "*configtest*"]*, "*delta*": "0:00:00.045046", "*end*": "2013-03-08 16:09:32.002063", "*rc*": 1, "*start*": "2013-03-08 16:09:31.957017"}
stderr: *Syntax error on line* 2 *of /etc/apache*2*/sites-enabled/awesome-app*:
Invalid command '*RocumentDoot*', *perhaps misspelled or defined by a module not included in the server configuration*
stdout: *Action* '*configtest*' *failed*.
The Apache error log may have more information.
FATAL: *all hosts have already failed -- aborting*
PLAY RECAP **********
host1.example.org : ok=4 changed=2 unreachable=0 failed=1

Как вы заметили, *apache*2*ctl* возвращает код ошибки 1. *Ansible* видит это и останавливает работу.

Наш виртуальный хост все равно был добавлен. При любой последующей попытке перезапуска *Apache* будет «ругаться» на конфигурацию и выключаться. Так что нам нужен способ отлавливать ошибки и возвращаться к рабочему состоянию.

Использование условий. Мы установили *Apache*, добавили виртуальный хост и перезапустили сервер. Но мы хотим вернуться к рабочему состоянию, если что-то пошло не так.

Возврат при проблемах. Прошлая ошибка – не вина *Ansible*. Это не система резервного копирования, и она не умеет откатывать все к прошлым состояниям. Безопасность плейбуков – ваша ответственность. *Ansible* просто не знает, как отменить эффект *a*2*ensite awesome-app*.

Как было изложено выше, если задача не может исполниться – обработка останавливается, но мы можем принять ошибку (и нам нужно это делать) [14]. Так мы и поступим: продолжим обработку в случае ошибки, но только чтобы вернуть все к рабочему состоянию:

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true
    - name: Push future default virtual host configuration
```

```
    copy: src=files/awesome-app dest=/etc/apache2/sites-available/
mode=0640
  - name: Activates our virtualhost
    command: a2ensite awesome-app
  - name: Check that our config is valid
    command: apache2ctl configtest
    register: result
    ignore_errors: True
  - name: Rolling back - Restoring old default virtualhost
    command: a2ensite default
    when: result|failed
  - name: Rolling back - Removing our virtualhost
    command: a2dissite awesome-app
    when: result|failed
  - name: Rolling back - Ending playbook
    fail: msg="Configuration file is not valid. Please check that be-
fore re-running the playbook."
    when: result|failed
  - name: Deactivates the default virtualhost
    command: a2dissite default
  - name: Deactivates the default ssl virtualhost
    command: a2dissite default-ssl
  notify:
    - restart apache
 handlers:
  - name: restart apache
    service: name=apache2 state=restarted
```

Ключевое слово *register* записывает вывод команды *apache2ctl configtest* (*exitstatus*, *stdout*, *stderr*, ...), и *when*: *result|failed* проверяет, содержит ли переменная(*result*) статус *failed*.

***Запускаем*:**

```
$ ansible-playbook -i step-07/hosts -l host1.example.org step-
07/apache.yml
PLAY [web] ********************
GATHERING FACTS ********************
ok: [host1.example.org]
TASK: [Installs apache web server] ********************
ok: [host1.example.org]
```

TASK: [Push future default virtual host configuration] **********

ok: *[host*1*.example.org]*

TASK: *[Activates our virtualhost]* **********

changed: [host1.example.org]

TASK: [Check that our config is valid] **********

failed: *[host*1*.example.org]* => {"*changed*": *true*, "*cmd*": *["apache*2*ctl*", "*configtest*"], "*delta*": "0:00:00.051874", "*end*": "2013-03-10 10:50:17.714105", "*rc*": 1, "*start*": "2013-03-10 10:50:17.662231"}

stderr: *Syntax error on line* 2 *of /etc/apache*2*/sites-enabled/awesome-app*:

Invalid command 'RocumentDoot', perhaps misspelled or defined by a module not included in the server configuration

stdout: *Action 'configtest' failed.*

The Apache error log may have more information.

...ignoring

TASK: *[Rolling back - Restoring old default virtualhost]* **********

changed: [host1.example.org]

TASK: [Rolling back - Removing our virtualhost] **********

changed: *[host*1*.example.org]*

TASK: *[Rolling back - Ending playbook]* **********

failed: [host1.example.org] => {"failed": true}

msg: Configuration file is not valid. Please check that before re-running the playbook.

FATAL: all hosts have already failed -- aborting

PLAY RECAP **********

*host*1*.example.org* : *ok*=7 *changed*=4 *unreachable*=0 *failed*=1

Кажется, все работает как нужно. Давайте попробуем перезапустить *apache*:

$ *ansible -i step*-07*/hosts -m service -a 'name=apache*2 *state=restarted' host*1*.example.org*

*host*1*.example.org | success >>* {

"*changed*": *true*,

"*name*": "*apache*2",

"*state*": "*started*"

}

Теперь наш *Apache* защищен от ошибок конфигурации. Помните, переменными можно пользоваться практически везде, так что этот плейбук можно использовать для *apache* и в других случаях. Напишите один раз и пользуйтесь везде.

Деплоим сайт с помощью Git. Мы установили *Apache*, добавили виртуальный хост и безопасно перезапустили сервер. Теперь давайте используем модуль *git*, чтобы сделать деплой приложения.

Модуль git. Модуль *git* – это просто еще один модуль. Но давайте попробуем что-нибудь интересное. А позже это пригодится, когда мы будем работать с *ansible-pull*.

Виртуальный хост задан, но нам нужно внести пару изменений чтобы закончить деплой. Мы деплоим приложение на *PHP*, так что нужно установить пакет *libapache2-mod-php5*. Также нужно установить сам *git*, так как, очевидно, модуль *git* требует его наличия.

Можно сделать так:

```
...
- name: Installs apache web server
  apt: pkg=apache2 state=installed update_cache=true
- name: Installs php5 module
  apt: pkg=libapache2-mod-php5 state=installed
- name: Installs git
  apt: pkg=git state=installed
...
```

но в *Ansible* есть способ лучше. Он может проходить по набору элементов и использовать каждый в определенном действии, вот так:

```
- hosts: web
  tasks:
    - name: Updates apt cache
      apt: update_cache=true
    - name: Installs necessary packages
      apt: pkg={{ item }} state=latest
      with_items:
        - apache2
        - libapache2-mod-php5
        - git
    - name: Push future default virtual host configuration
      copy:  src=files/awesome-app  dest=/etc/apache2/sites-available/
mode=0640
    - name: Activates our virtualhost
```

```
    command: a2ensite awesome-app
  - name: Check that our config is valid
    command: apache2ctl configtest
    register: result
    ignore_errors: True
  - name: Rolling back - Restoring old default virtualhost
    command: a2ensite default
    when: result|failed
  - name: Rolling back - Removing out virtualhost
    command: a2dissite awesome-app
    when: result|failed
  - name: Rolling back - Ending playbook
    fail: msg="Configuration file is not valid. Please check that before re-running the playbook."
    when: result|failed
  - name: Deploy our awesome application
    git: repo=https://github.com/leucos/ansible-tuto-demosite.git dest=/var/www/awesome-app
    tags: deploy
  - name: Deactivates the default virtualhost
    command: a2dissite default
  - name: Deactivates the default ssl virtualhost
    command: a2dissite default-ssl
    notify:
      - restart apache
 handlers:
  - name: restart apache
    service: name=apache2 state=restarted
```

***Запускаем*:**

```
$ ansible-playbook -i step-08/hosts -l host1.example.org step-08/apache.yml
PLAY [web] ********************
GATHERING FACTS ********************
ok: [host1.example.org]
TASK: [Updates apt cache] ********************
ok: [host1.example.org]
TASK: [Installs necessary packages] ********************
changed: [host1.example.org] => (item=apache2,libapache2-mod-php5,git)
```

TASK: *[Push future default virtual host configuration]* **********

changed: [host1.example.org]

TASK: [Activates our virtualhost] **********

changed: *[host*1*.example.org]*

TASK: *[Check that our config is valid]* **********

changed: [host1.example.org]

TASK: [Rolling back - Restoring old default virtualhost] **********

skipping: *[host*1*.example.org]*

TASK: *[Rolling back - Removing out virtualhost]* **********

skipping: [host1.example.org]

TASK: [Rolling back - Ending playbook] **********

skipping: *[host*1*.example.org]*

TASK: *[Deploy our awesome application]* **********

changed: [host1.example.org]

TASK: [Deactivates the default virtualhost] **********

changed: *[host*1*.example.org]*

TASK: *[Deactivates the default ssl virtualhost]* **********

changed: [host1.example.org]

NOTIFIED: [restart apache] **********

changed: *[host*1*.example.org]*

PLAY RECAP **********

host1.example.org : ok=10 changed=8 unreachable=0 failed=0

Теперь можно перейти на *http*://192.168.33.11 и увидеть «котенка» и имя сервера.

Строка *tags*: *deploy* позволяет запустить определнную порцию плейбука. Допустим, вы запушили новую версию сайта. Вы хотите ускорить процесс и запустить только ту часть, которая ответственна за деплой. Это можно сделать с помощью тегов. Естественно, «*deploy*» – это просто строка, можно задавать любую. Давайте посмотрим, как это можно использовать:

$ *ansible-playbook -i step*-08/*hosts -l host*1*.example.org step*-08/*apache.yml -t deployX*11 *forwarding request failed on channel* 0

PLAY [web] *****

*GATHERING FACTS *****ok*: *[host*1*.example.org]*

TASK: *[Deploy our awesome application] *****changed*: *[host*1*.example.org]*

*PLAY RECAP *****host*1*.example.org*: *ok*=2 *changed*=1 *unreachable*=0 *failed*=0

Добавляем еще один веб-сервер

У нас есть один веб-сервер. Мы хотим два.

Обновление inventory. Мы ожидаем наплыва трафика, так что давайте добавим еще один веб-сервер и балансировщик, который мы настроим в следующем шаге. Давайте закончим с *inventory*:

[web]

*host*1*.example.org ansible_ssh_host*=192.168.33.11 *ansible_ssh_user=root*

*host*2*.example.org ansible_ssh_host*=192.168.33.12 *ansible_ssh_user=root*

[haproxy]

*host*0*.example.org ansible_ssh_host*=192.168.33.10 *ansible_ssh_user=root*

Помните, здесь мы указываем *ansible_ssh_host*, потому что хост имеет не тот *IP*, что ожидается. Можно добавить эти хосты к себе в */etc*/*hosts* или использовать реальные имена (что вы и будете делать в обычной ситуации).

Сборка второго веб-сервера. Деплой второго сервера очень прост:

$ *ansible-playbook -i step*-09/*hosts step*-09/*apache.yml*

*PLAY [web] *********************

*GATHERING FACTS *********************

ok: *[host*2*.example.org]*

ok: *[host*1*.example.org]*

TASK: *[Updates apt cache] *********************

ok: [host1.example.org]

ok: [host2.example.org]

*TASK: [Installs necessary packages] *********************

ok: *[host*1*.example.org]* => (*item=apache*2*,libapache*2*-mod-php*5*,git*)

changed: *[host*2*.example.org]* => (*item=apache*2*,libapache*2*-mod-php*5*,git*)

TASK: [Push future default virtual host configuration] **********

ok: [host1.example.org]

changed: [host2.example.org]

TASK: [Activates our virtualhost] **********

changed: [host2.example.org]

changed: [host1.example.org]

TASK: [Check that our config is valid] **********

changed: [host2.example.org]

changed: [host1.example.org]

TASK: [Rolling back - Restoring old default virtualhost] **********

skipping: [host1.example.org]

skipping: [host2.example.org]

TASK: [Rolling back - Removing out virtualhost] **********

skipping: [host1.example.org]

skipping: [host2.example.org]

TASK: [Rolling back - Ending playbook] **********

skipping: [host1.example.org]

skipping: [host2.example.org]

TASK: [Deploy our awesome application] **********

ok: [host1.example.org]

changed: [host2.example.org]

TASK: [Deactivates the default virtualhost] **********

changed: [host1.example.org]

changed: [host2.example.org]

TASK: [Deactivates the default ssl virtualhost] **********

changed: [host2.example.org]

changed: [host1.example.org]

NOTIFIED: [restart apache] **********

changed: [host1.example.org]

changed: [host2.example.org]

PLAY RECAP **********

host1.example.org : ok=10 changed=5 unreachable=0 failed=0

host2.example.org : ok=10 changed=8 unreachable=0 failed=0

Все, что нужно, это удалить *-l host*1*.example.org* из командной строки. Помните, *-l* позволяет ограничить хосты для запуска. Теперь ограничения не требуется, и запуск произойдет на всех машинах группы *web*.

Если бы в группе *web* были другие машины, и нам нужно было бы запустить плейбук только на некоторых из них, можно было бы использовать, например, такое: *-l firsthost*:*secondhost*:....

Теперь у нас есть ферма веб-серверов, давайте превратим ее в кластер с помощью балансировщика нагрузок.

Шаблоны. Мы будем использовать *haproxy* в качестве балансировщика. Установка такая же, как с *apache*. Но конфигурация немного сложнее, потому что нам нужно указать список всех веб-серверов в конфигурации *haproxy*.

Шаблон конфигурации HAProxy. *Ansible* использует *Jinja*2, систему шаблонов для *Python*. Внутри *Jinja*2-шаблона можно использовать любую переменную, которая определена *Ansible*'ом.

Например, если нужно вывести на экран *inventory_name* хоста, для которого собран шаблон, то можно просто написать {{ *inventory_hostname* }} в *Jinja*2-шаблоне. Или, если нужно вывести *IP*-адрес первого *ethernet*-интерфейса (о котором *Ansible* знает благодаря модулю *setup*), то можно написать {{ *ansible_eth*1*['ipv*4*']['address'] }}.

*Jinja*2 также поддерживает условия, циклы и пр.

Давайте создадим директорию *templates*/ с *Jinja*-шаблоном внутри. Назовем его *haproxy.cfg.j*2. Расширение *.j*2 даем просто для удобства, оно не обязательно:

```
global
    daemon
    maxconn 256
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
listen cluster
    bind {{ ansible_eth1['ipv4']['address'] }}:80
    mode http
    stats enable
```

```
    balance roundrobin
{% for backend in groups['web'] %}
    server {{ hostvars[backend]['ansible_hostname'] }} {{ hostvars[backend]['ansible_eth1']['ipv4']['address'] }} check port 80
{% endfor %}
    option httpchk HEAD /index.php HTTP/1.0
```

Тут есть несколько новых для нас деталей.

Во-первых, {{ *ansible_eth*1*['ipv*4*']['address']* }} заменится на *IP* балансировщика нагрузки на *eth*1.

Дальше у нас есть цикл. Он используется для генерации списка бэкенд-серверов. Каждый шаг цикла соответствует одному хосту из группы *[web]*, и каждый такой хост будет записан в переменную *backend*. С помощью фактов хоста для каждого из хостов будет сгенерирована строка. Факты всех хостов доступны через переменную *hostvars*, поэтому достать переменные (например, имя хоста или *IP*, как в нашем случае) из других хостов очень легко.

Можно было написать список хостов вручную, у нас их всего два. Но мы надеемся, что популярность заставит нас заводить сотни серверов. Так что при добавлении или изменении серверов нам нужно лишь обновить группу *[web]*.

HAProxy playbook. Самое сложное позади. Написать плейбук для установки и конфигурации *HAproxy* очень легко:

```
- hosts: haproxy
  tasks:
    - name: Installs haproxy load balancer
      apt: pkg=haproxy state=installed update_cache=yes
    - name: Pushes configuration
      template: src=templates/haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg mode=0640 owner=root group=root
      notify:
        - restart haproxy

    - name: Sets default starting flag to 1
      lineinfile: dest=/etc/default/haproxy regexp="^ENABLED" line="ENABLED=1"
      notify:
        - restart haproxy
  handlers:
```

```
- name: restart haproxy
  service: name=haproxy state=restarted
```

Новый модуль тут только один: *template*. У него такие же аргументы, как у *copy*. А еще мы ограничили этот плейбук группой *haproxy*.

В нашем *inventory* содержатся только необходимые для кластера хосты, поэтому нам не нужно делать дополнительных ограничений и можно даже запустить оба плейбука. Но на самом деле нам нужно запускать их одновременно, так как *haproxy*-плейбуку нужны факты *из* двух веб-серверов. Чуть позже мы узнаем, как избежать этого:

```
$ ansible-playbook -i step-10/hosts step-10/apache.yml step-10/haproxy.yml
PLAY [web] *********************
GATHERING FACTS *********************
ok: [host1.example.org]
ok: [host2.example.org]
TASK: [Updates apt cache] *********************
ok: [host1.example.org]
ok: [host2.example.org]
TASK: [Installs necessary packages] **********************
ok: [host1.example.org] => (item=apache2,libapache2-mod-php5,git)
ok: [host2.example.org] => (item=apache2,libapache2-mod-php5,git)
TASK: [Push future default virtual host configuration] **********************
ok: [host2.example.org]
ok: [host1.example.org]
TASK: [Activates our virtualhost] **********************
changed: [host1.example.org]
changed: [host2.example.org]
TASK: [Check that our config is valid] **********************
changed: [host1.example.org]
changed: [host2.example.org]
TASK: [Rolling back - Restoring old default virtualhost] **********************
skipping: [host1.example.org]
skipping: [host2.example.org]
TASK: [Rolling back - Removing out virtualhost] **********************
skipping: [host1.example.org]
```

```
skipping: [host2.example.org]
TASK: [Rolling back - Ending playbook] **********
skipping: [host1.example.org]
skipping: [host2.example.org]
TASK: [Deploy our awesome application] **********
ok: [host2.example.org]
ok: [host1.example.org]
TASK: [Deactivates the default virtualhost] **********
changed: [host1.example.org]
changed: [host2.example.org]
TASK: [Deactivates the default ssl virtualhost] **********
changed: [host2.example.org]
changed: [host1.example.org]
NOTIFIED: [restart apache] **********
changed: [host2.example.org]
changed: [host1.example.org]
PLAY RECAP **********
host1.example.org : ok=10 changed=5 unreachable=0 failed=0
host2.example.org : ok=10 changed=5 unreachable=0 failed=0
PLAY [haproxy] **********
GATHERING FACTS **********
ok: [host0.example.org]
TASK: [Installs haproxy load balancer] **********
changed: [host0.example.org]
TASK: [Pushes configuration] **********
changed: [host0.example.org]

TASK: [Sets default starting flag to 1] **********
changed: [host0.example.org]
NOTIFIED: [restart haproxy] **********
changed: [host0.example.org]
PLAY RECAP **********
host0.example.org : ok=5 changed=4 unreachable=0 failed=0
```

Зайдите на *http*://192.168.33.10/ и оцените результат. Кластер задеплоен! Можно даже посмотреть на статистику *HAproxy*: *http*://192.168.33.10/*haproxy*?*stats*.

Снова переменные. Итак, мы установили балансировщик нагрузки, и он работает нормально. Мы берем переменные из фактов и используем их для генерации конфигурации.

Ansible также поддерживает другие виды переменных. Мы уже видели *ansible_ssh_host* в файле *inventory*, но теперь используем переменные, которые заданы в файлах *host_vars* и *group_vars*.

Тонкая настройка конфигурации HAProxy. Обычно *HAProxy* проверяет, живы ли бэкенды. Если бэкенд не откликается, то он удаляется из пула, и *HAProxy* больше не шлет ему запросы.

У бэкэндов может быть указан вес (от 0 до 256). Чем выше вес, тем больше запросов сервер получит по сравнению с другими серверами. Это полезно, когда узлы отличаются по мощности и нужно направить трафик в соответствии с этим.

Мы используем переменные для настройки этих параметров.

Group-переменные. Интервал проверки *haproxy* будет задан в файле *group_vars*. Таким образом, все экземпляры *haproxy* унаследуют это.

Нужно создать файл *group_vars*/*haproxy* внутри директории *inventory*. Название файла должно совпадать с названием группы, для которой задаются переменные. Если бы мы задавали переменные для группы *web*, то назвали бы файл *group_vars*/*web*:

haproxy_check_interval: 3000
haproxy_stats_socket: /*tmp*/*sock*

Название переменной может быть любым. Естественно, рекомендуется давать осмысленные названия, но каких-то специальных правил нет. Можно делать даже комплексные переменные (т. е. *Python dict*) вот так:

haproxy:
  check_interval: 3000
  stats_socket: /*tmp*/*sock*

Такой подход позволяет делать логическую группировку. Мы пока будем использовать простые переменные.

Переменные хоста. С переменными хоста такая же история, но файлы живут в директори *host_vars*. Давайте зададим вес бэкенда в *host_vars*/*host*1.*example.com*:

haproxy_backend_weight: 100

и для *host_vars*/*host*2.*example.com*:

haproxy_backend_weight: 150

Если бы мы задали *haproxy_backend_weight* в *group_vars*/*web*, то он бы использовался по-умолчанию: переменные из файла *host_vars* имеют приоритет перед переменными из *group_vars*.

Обновляем шаблон. Теперь необходимо обновить шаблон, чтобы он использовал эти переменные.

```
global
    daemon
    maxconn 256
{% if haproxy_stats_socket %}
    stats socket {{ haproxy_stats_socket }}
{% endif %}
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
listen cluster
    bind {{ ansible_eth1['ipv4']['address'] }}:80
    mode http
    stats enable
    balance roundrobin
{% for backend in groups['web'] %}
    server {{ hostvars[backend]['ansible_hostname'] }}
{{ hostvars[backend]['ansible_eth1']['ipv4']['address'] }}
check inter
{{ haproxy_check_interval }}
weight {{ hostvars[backend]['haproxy_backend_weight'] }} port 80
{% endfor %}
    option httpchk HEAD /index.php HTTP/1.0
```

Заметили блок {% *if* ...? Этот блок будет отработан, если условие верно. Так что, если мы где-нибудь зададим *haproxy_stats_socket* для балансировщика нагрузки (можно даже добавить --*extra-vars*="*haproxy_stats_sockets*=/*tmp*/*sock*" при вызове из командной строки), то блок будет добавлен в сгенерированный конфигурационный файл.

Имейте ввиду, что такой метод очень плох с точки зрения безопасности.

***Запускаем*:**

ansible-playbook -i step-11/*hosts step*-11/*haproxy.yml*

Мы можем, но не обязаны запускать плейбук *apache*, потому что ничего не изменилось. Но пришлось добавить небольшой «трюк». Вот обновленный плейбук *haproxy*:

```
- hosts: web
- hosts: haproxy
  tasks:
    - name: Installs haproxy load balancer
      apt: pkg=haproxy state=installed update_cache=yes
    - name: Pushes configuration
      template: src=templates/haproxy.cfg.j2
dest=/etc/haproxy/haproxy.cfg mode=0640 owner=root group=root
      notify:
        - restart haproxy
    - name: Sets default starting flag to 1
      lineinfile: dest=/etc/default/haproxy regexp="^ENABLED"
line="ENABLED=1"
      notify:
        - restart haproxy
  handlers:
    - name: restart haproxy
      service: name=haproxy state=restarted
```

Мы добавили пустой блок для веб-хостов в самом начале. В нем ничего не происходит. Но факт его наличия заставит *Ansible* собрать факты для группы *web*. Это необходимо, потому что плейбук *haproxy* использует факты из этой группы. Если не сделать этого, то *Ansible* будет «ругаться», что ключа *ansible_eth*1 не существует.

6.4. Роли в *Ansible* (*role*)

Теперь, когда все плейбуки готовы, давайте все отрефакторим. Мы переделаем все через роли. Роли – это просто еще один способ организации файлов, у них есть несколько интересных возможностей. Не будем вдаваться в детали, все они описаны в документации *Ansible*.

Зависимости ролей: роль *B* может зависеть от другой роли *A*. Поэтому при применении роли *B* автоматически будет применена роль *A*.

Структура ролей

Роли добавляют немного «магии» в *Ansible*: они предполагают особую организацию файлов. Роли полагается структурировать определенным образом, хотя вы можете делать это, как угодно вам. Тем не менее, если придерживаться соглашений, вам будет гораздо легче создавать модульные плейбуки. Содержать код в порядке будет гораздо легче. Рубисты называют это *convention over configuration*.

Структура файлов для ролей такая:

```
roles
 |
 |_some_role
    |
    |_files
    |  |
    |  |_file1
    |  |_...
    |
    |_templates
    |  |
    |  |_template1.j2
    |  |_...
    |
    |_tasks
    |  |
    |  |_main.yml
    |  |_some_other_file.yml
    |  |_ ...
    |
    |_handlers
    |  |
    |  |_main.yml
    |  |_some_other_file.yml
    |  |_ ...
    |
    |_vars
    |  |
```

```
|   |_main.yml
|   |_some_other_file.yml
|   |_ ...
|
|_meta
    |
    |_main.yml
    |_some_other_file.yml
    |_ ...
```

Это довольно просто.

Файлы *main.yml* не обязательны. Но если они присутствуют, то роли добавят их к отработке автоматически. Эти файлы можно использовать для добавления других тасков и хэндлеров.

Обратите внимание на директории *vars* и *meta*. *vars* нужна для случаев, когда есть куча переменных, связанных с ролью. Но мне лично не нравится задавать переменные в ролях и сценариях напрямую. Я считаю, что переменные должны быть частью конфигурации, а сценарии – это структура. Иными словами, я считаю сценарии фабриками, а данные – параметрами для фабрик. Поэтому я предпочитаю видеть «данные» (например, переменные) вне ролей и сценариев. Тогда мне легче распределять роли и не раскрывать слишком много информации о внутренностях серверов. Но это дело личных предпочтений. *Ansible* предоставляет вам выбор.

В директории *meta* находятся зависимости, но об этом поговорим в следующий раз. Сценарии лежат в директории *roles*.

Создаем роль Apache. Теперь у нас достаточно знаний, чтобы создать роль для *apache* на основе нашего плейбука.

Несколько простых шагов:

– создать директорию ролей и структуру роли *apache*;

– вынести хэндлер *apache* в файл *roles*/*apache*/*handlers*/*main.yml*;

– перенести конфигурационный файл *apache awesome-app* в *roles*/*apache*/*files*/;

– создать плейбук для роли.

***Задаем структуру*:**

mkdir -p step-12/*roles*/*apache*/{*tasks*,*handlers*,*files*}

Теперь копируем таски из *apache.yml* в *main.yml*. Файл выглядит так:

```
- name: Updates apt cache
  apt: update_cache=true
```

```
- name: Installs necessary packages
  apt: pkg={{ item }} state=latest
  with_items:
    - apache2
    - libapache2-mod-php5
    - git
...
- name: Deactivates the default ssl virtualhost
  command: a2dissite default-ssl
  notify:
    - restart apache
```

Это не полный текст файла, а просто иллюстрация. Файл в точности повторяет содержание *apache.yml* между *tasks* и *handlers*.

Мы также убрали обращения к директориям *files/* и *templates/* в тасках. Так как используется стандартная структура ролей, *Ansible* сам знает, в какие директории смотреть.

Выносим хэндлер. Нужно создать файл *step*-12/*roles*/*apache*/*handlers*/*main.yml*:

```
- name: restart apache
  service: name=apache2 state=restarted
```

Переносим файл конфигурации.

Еще проще:

```
cp step-11/files/awesome-app step-12/roles/apache/files/
```

Роль *apache* работает. Но нам нужен способ запустить ее.

Создаем плейбук роли. Давайте создадим плейбук верхнего уровня для связывания хостов и групп хостов с ролями. Назовем файл *site.yml*, так как нам нужна общая конфигурация сайта. Заодно добавим туда *haproxy*:

```
- hosts: web
  roles:
    - { role: apache }
- hosts: haproxy
  roles:
    - { role: haproxy }
```

Теперь давайте создадим роль *haproxy*:

```
mkdir -p step-12/roles/haproxy/{tasks,handlers,templates}
cp step-11/templates/haproxy.cfg.j2 step-12/roles/haproxy/templates/
```

потом извлечем хэндлер и удалим упоминание *templates*/.

ansible-playbook -i step-12/*hosts step*-12/*site.yml*

Если все хорошо, то мы увидим "*PLAY RECAP*":

*host*0.*example.org* : *ok*=5 *changed*=2 *unreachable*=0 *failed*=0
*host*1.*example.org* : *ok*=10 *changed*=5 *unreachable*=0 *failed*=0
*host*2.*example.org* : *ok*=10 *changed*=5 *unreachable*=0 *failed*=0

Вы, наверное, заметили, что запуск всех ролей в *site.yml* занимает много времени. Что если нужно сделать изменения только для веб-серверов? Используем *limit*-флаг:

ansible-playbook -i step-12/*hosts -l web step*-12/*site.yml*

На этом миграция на роли закончена.

Литература

1. Мейер, Б. Объектно-ориентированное конструирование и программирование систем : пер. с англ. / Б. Мейер. – М. : Рус. редакция, 2005. – 1232 с. : ил.
2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений : пер. с англ. / Г. Буч. – 3-е изд. – М. : Вильямс, 2008. – 720 с.
3. Мартин, Р. Быстрая разработка программ: принципы, примеры, практика : пер. с англ. / Р. Мартин. – М. : Вильямс, 2004. – 752 с.
4. Амблер, С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки / С. Амблер. – СПб. : Питер, 2005. – 412 с.
5. Фаулер, М. Архитектура корпоративных программных приложений. : пер. с англ. / М. Фаулер. – М. : Вильямс, 2004. – 544 с.
6. Вигерс, К. Разработка требований к программному обеспечению : пер. с англ. / К. Вигерс, Д. Битти. – 3-е изд, доп. – М. : Рус. редакция ; СПб. : БХВ-Петербург, 2018. – 736 с. : ил.
7. Брукс, Ф. Мифический человеко-месяц или как создаются программные системы / Ф. Брукс. – СПб. : Символ-Плюс, 2006. – 304 с. : ил.
8. Дюваль, М. Поль. Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска / М. Поль Дюваль. – М. : Вильямс, 2008. – 240 с.
9. Гецци, К. Основы инженерии программного обеспечения : пер. с англ. / К. Гецци, М. Джазаейри, Д. Мандриолн. – 2-е изд. – СПб. : БХВ-Петербург, 2005. – 832 с. : ил.
10. Благодатских, В. А. Стандартизация разработки программных средств : учеб. пособие / В. А. Благодатских, В. А. Волнин, К. Ф. Поскакалов ; под ред. О. С. Разумова. – М. : Финансы и статистика, 2006. – 288 с. : ил.
11. Котляров, В. П. Основы тестирования программного обеспечения / В. П. Котляров. – М. : ИНТУИТ, 2015. – 147 с.
12. Орлов, С. Технологии разработки программного обеспечения. Разработка сложных программных систем : учеб. пособие / С. Орлов. – СПб. : Питер, 2003. – 480 с. : ил.
13. Таейр, Т. Надежность программного обеспечения / Т. Таейр, М. Липов, Э. Нельсон. – М. : Мир, 1981. – 323 с.
14. Хамбл, Дж. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ : пер. с англ. / Дж. Хамбл, Д. Фарли. – М. : Вильямс, 2011. – 432 с. : ил.

Учебное издание

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ

Учебно-методическое пособие
для студентов специальностей
1-40 05 01 «Информационные системы и технологии
(по направлениям)» и 1-40 80 04 «Информатика
и технологии программирования»
дневной и заочной форм обучения

Составители: **Стефановский** Игорь Леонидович
Соболев Денис Викторович

Редактор *Т. Н. Мисюрова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 05.04.22.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 11,62. Уч.-изд. л. 12,82.
Тираж 30 экз. Заказ № 173/21.

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель