Непрерывная интеграция

улучшение качества программного обеспечения и снижение риска

Continuous Integration

Improving Software Quality and Reducing Risk

Paul M. Duvall
with
Steve Matyas and Andrew Glover

ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokio • Singapore • Mexico City

Непрерывная интеграция

улучшение качества программного обеспечения и снижение риска

Поль М. Дюваль,
а также
Стивен Матиас и Эндрю Гловер

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
Д95
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.А. Коваленко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Дюваль, Поль М., **Матиас III**, Стивен М., **Гловер**, Эндрю.

Д95 Непрерывная интеграция: улучшение качества программного обеспечения и снижение риска. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 240 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1408-8 (рус.)

В этой книге рассматриваются некоторые из наиболее типичных процессов разработки программного обеспечения: компиляция кода, определение данных и манипулирование ими в базе данных; осуществление проверки, просмотр кода и в конечном итоге развертывание программного обеспечения. Но главное, в ней описано, как непрерывная интеграция способна снизить риски, которые подстерегают при создании приложений. В системе непрерывной интеграции большинство этих процессов автоматизировано, и они запускаются после каждого изменения разрабатываемого программного обеспечения.

В книге обсуждаются аспекты автоматизации непрерывной интеграции, большинство предоставляемых ей преимуществ в области повторяемых и склонных к ошибкам процессов. Ныне существует множество великолепных инструментальных средств, поддерживающих непрерывную интеграцию как автоматизированный процесс, использующий сервер CI для автоматизации действий. Тем не менее ручной подход к интеграции (при автоматизированной компиляции) вполне может хорошо сработать.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-1408-8 (рус.)
ISBN 0-321-33638-0 (англ.)

Оглавление

Содержание

Об авторах

Поль М. Дюваль (Paul M. Duvall) — главный технический директор консалтинговой фирмы Stelligent Incorporated, а также интеллектуальный лидер групп помощи по быстрой и надежной разработке программного обеспечения, его усовершенствования и оптимизации процесса создания. Он побывал в роли практически каждого участника проекта программного обеспечения, от разработчика и испытателя архитектуры до руководителя проекта. Поль консультировал клиентов в различных отраслях, включая финансы, недвижимость, правительство, здравоохранение и множество независимых производителей. Его часто приглашают на многие известные конференции по программному обеспечению в качестве почетного председателя. Поль написал книгу *Automation for the People* из серии *IBM developerWorks*, а также является соавтором книг *NFJS 2007 Anthology* (издательство Pragmatic Programmers, 2007) и *UML 2 Toolkit* (издательство Wiley, 2003). Еще он соавтор системы управления данными клинических исследований, а также метода устойчивости к задержкам. Его блоги доступны на сайтах `www.testearly.com` и `www.integratebutton.com`.

Стивен М. Матиас III (Stephen M. Matyas III) — вице-президент компании AutomateIT, отделения службы 5AM Solutions, Inc., которая помогает организациям в улучшении разработки программного обеспечения за счет автоматизации. Стив имеет обширный опыт по созданию прикладного программного обеспечения, включая работу как с коммерческими, так и с правительственными заказчиками. Стив занимал множество различных должностей, от бизнес-аналитика и руководителя проекта до разработчика, дизайнера и архитектора. Еще он является соавтором книги *UML 2 Toolkit* (издательство Wiley, 2003). Стивен на практике применяет многие из итерационных и инкрементных методов разработки, включая Agile и Rational Unified Process (RUP). Он обладает огромным профессиональным опытом в области разработки специального программного обеспечения с использованием Java/J2EE, причем специализируется на методиках, качестве программного обеспечения и усовершенствовании процесса разработки. Стивену присвоена степень бакалавра наук по информатике политехнического института штата Вирджиния, а также государственного университета.

Эндрю Гловер (Andrew Glover) — президент консалтинговой фирмы Stelligent Incorporated, интеллектуальный лидер групп помощи по быстрой и надежной разработке программного обеспечения, его улучшения и оптимизации процесса создания. Энди часто выступает на различных конференциях, а также на симпозиуме No Fluff Just Stuff Software Symposium; кроме того, он соавтор книг *Groovy in Action* (издательство Manning, 2007), *Java Testing Patterns* (издательство Wiley, 2004) и *NFJS 2006 Anthology* (серия *Pragmatic Programmers*, 2006). Он также автор различных сетевых публикаций, включая такие порталы, как developerWorks от IBM, ONJava и Dev2Dev. Его блоги о качестве программного обеспечения доступны на сайтах `www.thediscoblog.com` и `www.testearly.com`.

О соавторах

Лайза Портер (Lisa Porter) — старший технический разработчик для консалтинговых групп, предоставляющих решения по сетевой безопасности для правительства США. До этой книги она занималась техническим редактированием. На ее счету участие в многочисленных больших проектах по разработке программного обеспечения, позволивших ей приобрести богатый опыт в определении требований, а также возможностей реализации проекта. Она также применила принципы технического редактирования в области перевода на другие языки, проектирования и архитектуры. Лайза редактирует книги и сетевые публикации начиная с 2002 года.

Эрик Тавела (Eric Tavela) — главный архитектор компании по разработке программного обеспечения 5AM Solutions, Inc., специализирующейся на внедрении новейших технологий, а также поддержке научных исследований. Основной специализацией Эрика является проектирование и реализация приложений Java/J2EE, а также обучение разработчиков методам объектно-ориентированного программирования и моделирования UML.

Господь благословил меня замечательной семьей.
Посвящается моим родителям, Полю и Ноне, моим братьям и сестрам, Сью, Джоан, Джону, Мэри, Салли, Тиму, Полин и Эви.
П.М.Д.

Введение

Предисловие Мартина Фаулера[1]

В прежние времена одним из наиболее сложных и напряженных моментов проектирования программного обеспечения была интеграция. При сборке воедино модулей, разработанных по отдельности, зачастую возникают проблемы, обнаружить причины которых бывает крайне сложно. Однако за последние несколько лет интеграция почти перестала быть головной болью проекта, во всяком случае теперь это уже *не событие*.

Причиной такого превращения является приобретение навыков интеграции. С одной стороны, ежедневное построение можно считать амбициозной целью, с другой — в большинстве известных мне проектов интеграция осуществлялась по несколько раз в день. Как ни парадоксально, но когда встречается некое затруднительное действие, хороший совет — делать его почаще.

Самым интересным в непрерывной интеграции является то, что людей зачастую удивляет ее влияние. Нередко люди отказываются от нее как от маргинальной возможности, хотя это может полностью изменить стиль проекта. Однако смысл в этом намного бо́льший, ведь так проблемы обнаруживаются быстрее. Поскольку времени между возникновением и обнаружением ошибки проходит меньше, ее проще найти, тем более что можно легко просмотреть сделанные изменения и отыскать источник проблем. Вместе с соответствующей программой проверки это может привести к существенному уменьшению количества ошибок. В результате разработчики тратят меньше времени на отладку, больше занимаются реализацией возможностей и сохраняют уверенность в надежности создаваемого.

Безусловно, простого совета интегрировать чаще недостаточно, поскольку за этим кроется набор принципов и практический опыт, позволяющие сделать непрерывную интеграцию действительностью. Многое из упомянутого здесь можно найти рассеянным по другим книгам и Интернету (и я горд, что принял участие в создании этого содержимого), но все это придется искать самостоятельно.

Поэтому я очень рад, что Поль собрал всю эту информацию в одну книгу, настоящий справочник для желающих получить полезный совет. Подобно любой практике наибольшие проблемы кроются в деталях. За последние несколько лет мы многое узнали об этих составляющих и научились справляться с проблемами. Эта книга обобщает накопленный опыт, предоставляя надежную основу знаний о непрерывной интеграции, а также раскрывает ее значение для разработки программного обеспечения.

Предисловие Пола Джулиуса

Я был уверен, что рано или поздно кто-нибудь напишет подобную книгу. По секрету, я всегда надеялся, что это буду я. Но я доволен, что Поль, Стив и Энди наконец связали все это воедино в продуманный трактат.

Я всегда занимался чем-то похожим на непрерывную интеграцию. В марте 2001 года я участвовал в основании проекта с открытым исходным кодом CruiseControl, будучи его администратором. В моей повседневной работе я постоянно консультировался с консал-

[1] Мартин Фаулер — редактор серии и главный научный сотрудник компании ThoughtWorks.

тинговой компанией ThoughtWorks, помогающей в создании клиентских структур, построении и развертывании решений проверки с использованием принципов и инструментов непрерывной интеграции.

Работая с почтовыми сообщениями проекта CruiseControl до 2003 года, я получил возможность прочитать описания тысяч различных случаев непрерывной интеграции. Проблемы, с которыми сталкиваются создатели программного обеспечения, весьма разнообразны и сложны. Со временем я все яснее видел причины, по которым разработчики мирились с этими сложностями. Преимущества непрерывной интеграции, такие как быстрая обратная связь, быстрое развертывание и воспроизводимая автоматизированная проверка, существенно их превышали. Однако как просто зачастую пропустить метку при создании типов окружения. Я никогда не предполагал, когда мы впервые выпустили CruiseControl, насколько захватывающи способы, которыми будет использоватся непрерывная интеграция для улучшения процесса разработки программного обеспечения.

В 2000 году я работал над большим проектом приложения J2EE, используя все возможности, предоставляемые данной спецификацией. Приложение было в своем роде уникальным, но создавалось сложно. Под построением я подразумеваю компиляцию, проверку, архивирование и проверку функций. Система Ant все еще пребывала в младенчестве и еще не стала фактическим стандартом для приложений Java. Для компиляции всего необходимого и запуска проверки модулей мы просто использовали ряд сценариев оболочки. Для передачи всего в разворачиваемый архив применялся еще один набор сценариев оболочки. И наконец, мы вручную выполняли ряд операций по развертыванию файлов JAR и запуску комплекта проверки функций. Само собой разумеется, этот процесс был трудоемок, утомителен и чреват ошибками.

Так начались мои поиски перенастраиваемого средства "построения", для которого было бы достаточно нажать "одну кнопку" (одна из любимых тем Мартина Фаулера). Система Ant решила проблему создания независимых от платформы сценариев построения. Оставалось найти что-нибудь для выполнения таких утомительных действий, как развертывание, проверка функций и сообщение о результатах. Я исследовал существующие решения, но напрасно. Работая над тем проектом, я так и не нашел достаточно работоспособного средства, которое подходило бы мне полностью. Разработка приложения завершилась успешно, но я знал, что все можно было сделать проще и лучше.

Ответ я отыскал между завершением того проекта и началом следующего. Мартин Фаулер и Матт Фоеммел (Matt Foemmel) как раз опубликовали свою фундаментальную статью о непрерывной интеграции. Случайно я познакомился с людьми из ThoughtWorkers, работавшими над реализацией системы многократного использования решения Фаулера и Фоеммела. Я был несказанно рад! Я знал, что это ответ на все мои вопросы, оставшиеся от предыдущего проекта. Через несколько недель у нас уже все было готово, и мы применили данный подход в нескольких существующих проектах. Я даже посетил сайт бета-тестирования предшественника системы CruiseControl, чтобы опробовать ее в полном объеме. Вскоре после этого мы перешли на реализацию с открытым исходным кодом. Для меня уже не было никакого смысла оглядываться назад.

В качестве консультанта в ThoughtWorks я сталкивался с некоторыми из наиболее сложных случаев развертывания корпоративных архитектур. Нашим клиентам зачастую требовалось быстрое решение на основании преимуществ высокоуровневых концепций, обещанных в литературе. Подобно любой технологии, в них есть приличная доза дезинформации о том, насколько легко все это внедрить на предприятии. Если годы консультаций и научили меня чему, так это тому, что ничто не является столь же простым, как кажется.

Мне нравится разговаривать с клиентами о реальном применении принципов непрерывной интеграции. Я предпочитаю подчеркнуть важность перевода разработки "интонацией", чтобы корректно обозначить преимущества. Когда разработчики проводят проверки только раз в месяц, не уделяя внимания автоматизированному контролю, или не имеют никакой заинтересованности в немедленном устранении ошибок, их ожидают большие неприятности, для решения которых понадобятся все преимущества непрерывной интеграции.

Означает ли это, что менеджеры IT не должны ничего знать о непрерывной интеграции до тех пор, пока дело не дойдет до проблем? Вовсе нет. Фактически практическое применение непрерывной интеграции может быть одним из наиболее существенных мотивов для изменений. Я полагаю, что установка таких инструментов непрерывной интеграции, как CruiseControl, мотивирует активность группы разработчиков программного обеспечения. Изменение не происходит внезапно, и вы должны соразмерить ваши ожидания соответственно (это относится и к менеджерам IT). При настойчивости и хорошем понимании основных принципов даже наиболее сложные системы могут быть созданы простыми и понятными, легкими для проверки и быстрой разработки.

Этой книгой авторы подготовили хорошее игровое поле. Я полагаю, что она будет не только исчерпывающей, но и весьма популярной долгое время. Глубокий анализ наиболее важных аспектов непрерывной интеграции поможет читателям принимать хорошо продуманные решения. Диапазон рассматриваемых тем обширен и включает описание доминирующих ныне подходов непрерывной интеграции, помогая читателям взвесить компромиссы, на которые они должны будут пойти. И наконец, мне нравится наблюдать, как задачи, к достижению которых стремятся столь многие в сообществе непрерывной интеграции, постепенно формализуются и становятся основой для дальнейших исследований. Поэтому я настоятельно рекомендую эту книгу как жизненно важный ресурс для разработчиков корпоративных приложений в условиях сложной географии, использующих магию непрерывной интеграции.

Предисловие

В начале моей карьеры в одном из журналов я увидел анонс на всю страницу, где был изображен фрагмент клавиатуры с клавишей, подобной клавише <Enter> и помеченной словом "Integrate" (Интегрировать) (рис. П.1). Текст ниже гласил: "Если бы только это было так просто". Не помню, кто и для чего привел эту иллюстрацию, но мысль мне понравилась. Однако в свете разработки программного обеспечения я полагал, что такое никогда не будет возможным, поскольку в нашем проекте мы провели несколько дней в "аду интеграции", пытаясь собрать воедино бесчисленные компоненты программного обеспечения в конце каждого промежуточного этапа проекта. Однако концепция мне понравилась настолько, что я вырезал рисунок и повесил его у себя на стенке. Одной из главных задач эффективной разработки программного обеспечения мне представлялась автоматизация повторяемых процессов и процессов, склонных к ошибкам. Кроме того, это воплощало мое убеждение в возможности сделать интеграцию программного обеспечения "не событием" проекта (как это назвал Мартин Фаулер), а неким обычным действием. *Непрерывная интеграция* (Continuous Integration — CI) способна превратить это в реальность.

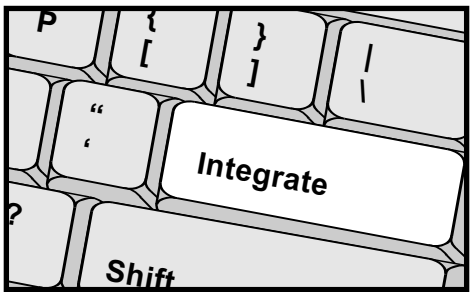


Рис. П.1 Кнопка <Integrate>!

О чем эта книга

Здесь описаны некоторые из наиболее типичных процессов разработки программного обеспечения: компиляция кода, определение данных и манипулирование ими в базе данных; осуществление проверки, просмотр кода и, в конечном счете, развертывание программного обеспечения. Кроме того, участники группы, безусловно, должны общаться друг с другом при обсуждении состояния программного обеспечения. Только представьте, что все эти процессы можно запустить, нажав всего лишь одну кнопку.

В книге рассматривается создание виртуальной кнопки <Integrate>, автоматизирующей большинство процессов разработки программного обеспечения. Но главное, в ней описано, как такая кнопка способна снизить риски, подстерегающие при построении приложений, например, от позднего обнаружения дефектов и низкокачественного кода. В системе CI большинство этих процессов автоматизировано, и они запускаются после каждого изменения разрабатываемого программного обеспечения.

Что такое непрерывная интеграция?

Процесс интеграции программного обеспечения — далеко не новая проблема. В проекте, выполняемом одним человеком с немногими внешними зависимостями, интеграция программного обеспечения — не слишком существенная проблема, но при увеличении сложности проекта (даже если в него просто добавлен еще один человек) возникает насущная потребность в интеграции и проверке слаженной работы компонентов программного обеспечения, причем заранее и *часто*. Дожидаться конца проекта для проведения интеграции и выявления всего спектра возможных ошибок — неразумно и к тому же не способствует качеству программного обеспечения, а зачастую даже приводит к удорожанию и задержке сдачи проекта. Непрерывная интеграция снижает подобные риски.

В своей популярной статье “Continuous Integration”[2] Мартин Фаулер описывает CI так:

... практика разработки программного обеспечения, когда участники группы осуществляют частую интеграцию своих работ. Обычно каждый человек проводит интеграцию по крайней мере ежедневно, что приводит к нескольким интеграциям в день. Для максимально быстрого обнаружения ошибок каждая интеграция осуществляется автоматизированно (вместе с проверкой). Многие группы находят, что данный подход позволяет значительно уменьшить проблемы интеграции и способствует более быстрой разработке программного обеспечения.

[2] См. `www.martinfowler.com/articles/continuousIntegration.html`.

А это означает следующее:

- все разработчики выполняют закрытое построение[3] на собственных рабочих станциях перед передачей кода в хранилище с контролем версий, для гарантии того, что внесенные изменения не приведут к ошибке при интеграционном построении;
- разработчики обновляют свой код в хранилище с контролем версий *по крайней мере* один раз в день;
- интеграционное построение осуществляется несколько раз в день на выделенной для этого машине;
- при каждом построении проводится 100 % проверок;
- создаваемый продукт (например, файл WAR, сборка, исполняемый файл и т.д.) пригоден для функциональной проверки;
- исправление ошибок имеет самый высокий приоритет;
- часть разработчиков просматривают отчеты, созданные в ходе построения, стандарты программирования и отчеты анализа зависимостей, в поисках областей для усовершенствования.

В этой книге обсуждаются аспекты автоматизации CI, большинство предоставляемых ей преимуществ в области повторяемых и склонных к ошибкам процессов; однако Фаулер идентифицирует CI как процесс часто осуществляемый, но не как автоматизированный. Однако, поскольку ныне существует множество великолепных инструментальных средств, поддерживающих CI как автоматизированный процесс, использующий сервер CI для автоматизации действий CI — данный подход эффективен. Тем не менее ручной подход к интеграции (при автоматизированной компиляции) вполне может хорошо сработать в любой группе.

Быстрая обратная связь

Непрерывная интеграция увеличивает возможности обратной связи. Она позволяет следить за состоянием проекта в течение дня. CI применяется для уменьшения временного промежутка между моментом проявления дефекта и его устранением, улучшая таким образом общее качество программного обеспечения.

Группе разработки вовсе не следует полагать, что автоматизация системы CI избавила их от проблем интеграции. Это тем более справедливо, если группа использует автоматизированный инструмент не более чем для компиляции исходного кода; некоторые именно это считают “построением” (build), что совсем не так (см. главу 1, “Первые шаги”). Эффективная практика CI подразумевает намного больше, чем применение соответствующих инструментов. Сюда относятся действия, которые будут описаны в этой книге, например частые обновления файлов в хранилище с контролем версий, немедленное устранение проблем, а также использование отдельной машины для интеграционного построения.

Практика CI обеспечивает более быструю обратную связь. Применение эффективных практик CI позволяет узнавать общее состояние разрабатываемого программного обеспечения *по нескольку раз в день*. Более того, CI хорошо работает с такими практиками, как рефакторинг и разработка методом проверки, поскольку в их основе лежит концепция небольших изменений. В сущности, CI гарантирует совместимость последних с остальной

[3] Схемы закрытого (системного) и интеграционного построения описаны в книге *Software Configuration Management Patterns* Стивена П. Беркзука (Stephen P. Berczuk) и Бреда Апплетона (Brad Appleton).

частью программного обеспечения. На более высоком уровне CI повышает коллективную ответственность группы и снижает трудоемкость проекта, поскольку уменьшает объем *ручного* труда, выполняемого при каждом изменении разрабатываемого программного обеспечения.

Замечание о слове "непрерывное"

В этой книге используется термин "непрерывная" ("continuous"), однако его употребление технически неправильно. *Непрерывность* подразумевает нечто, что, будучи начато один раз, никогда не завершается. Это предполагает непрерывность процесса интеграции, а этого нет даже в наиболее интенсивной системе CI. Таким образом, в этой книге описывается то, что скорее является "очень частой интеграцией".

Для кого написана эта книга

Опыт свидетельствует, что существует отличие между теми, кто рассматривает разработку программного обеспечения как *задачу*, и теми, кто считает ее своей *профессией*. Эта книга для тех, кто работает профессионально и выполняет в проекте повторяемые процессы. В ней описываются практики и преимущества CI, а также предоставляется информация о том, как их применять для высвобождения времени и накопления опыта для решения более важных проблем.

Данная книга раскрывает основные темы, связанные с CI, включая реализацию последней с использованием непрерывной обратной связи, проверки, развертывания, инспекции и интеграции базы данных. Независимо от того, какова роль читателя в разработке программного обеспечения, можно включать элементы CI в собственные процессы построения. Если вы профессионал-разработчик, желающий повысить эффективность своей работы, т.е. делать больше за то же время и с более предсказуемым результатом, то эта книга для вас.

Разработчики

Если вы заметили, что вместо разработки программного обеспечения для пользователей занимаетесь решением проблем интеграции, то эта книга поможет избавиться от них и вернуться к основной работе. Настоящая книга не требует больших временны*х* затрат на интеграцию; она, наоборот, о том, как облегчить интеграцию программного обеспечения и позволить сосредоточиться на любимом занятии — разработке. Многие практики и примеры, приведенные в книге, демонстрируют способы реализации эффективной системы CI.

Управление построением, настройкой и выпуском

Если ваша задача заключается в выпуске *работоспособного* программного обеспечения, то вы найдете эту книгу особенно интересной, поскольку в ней демонстрируется, что запуск процессов при *каждом* внесении изменений в хранилище с контролем версий позволит создавать слаженное работоспособное программное обеспечение. Большинство руководителей проекта выполняют и другие функции в процессе построения, например разработку. Система CI сможет самостоятельно решить некоторые из подобных задач, тем самым избавив от необходимости ждать конца цикла разработки, чтобы получить развертываемое и *проверяемое* (testable) программное обеспечение.

Испытатели

Непрерывная интеграция обеспечивает быструю обратную связь при разработке программного обеспечения, но не устраняет традиционных повторно встречающихся дефектов даже после запуска "фиксации". Испытатели обычно получают удовлетворение от своей роли в проекте, где применяется CI, поскольку программное обеспечение чаще доступно для проверки, к тому же с меньшими проверяемыми областями. При использовании системы CI в цикле разработки контролируется *все и всегда*, в отличие от традиционного сценария, где испытатели либо проверяют все в последние часы или не проверяют вовсе.

Менеджеры

Эта книга может произвести на вас впечатление, если вы ищете высоко компетентное издание для группы, регулярно и неоднократно выпускающей работоспособное программное обеспечение. Вы можете контролировать сроки, бюджет и качество куда эффективнее, поскольку будете основывать свои решения по рабочему программному обеспечению на фактической обратной связи и показателях, а не только на элементах задачи и расписании проекта.

Структура книги

Данная книга разделена на две части. Первая содержит введение в CI, исследующее ее концепции и практики. Эта часть адресована тем читателям, которые не знакомы с базовыми практиками CI. Хотя, безусловно, на этом практики CI не исчерпываются. Вторая часть развертывает базовые концепции в набор эффективных процессов, выполняемых системами CI, такими как проверка, инспекция, развертывание и обратная связь.

Часть I. Основы CI — принципы и практики

Глава 1, "Первые шаги", сразу вводит в курс дела высокоуровневым примером использования сервера CI при непрерывном построении программного обеспечения.

Глава 2, "Введение в непрерывную интеграцию", знакомит с общими практиками и способами применения CI.

Глава 3, "Снижение риска с использованием CI", обсуждает ключевые риски, которые может смягчать CI, на примере использования сценариев.

Глава 4, "Построение программного обеспечения при каждом изменении", исследует практику интегрирования программного обеспечения при каждом изменении с применением автоматизированной компиляции.

Часть II. Создание полнофункциональной системы CI

Глава 5, "Непрерывная интеграция баз данных", переходит к более сложным концепциям, включая процесс перестройки базы данных и применения проверки последних в качестве частей интеграционного построения.

Глава 6, "Непрерывная проверка", рассматривает концепции и стратегии проверки программного обеспечения при каждом интеграционном построении.

Глава 7, "Непрерывная инспекция", знакомит с использованием различных инструментальных средств и методов автоматизированной, а также непрерывной инспекции (статический и динамический анализ).

Глава 8, “Непрерывное развертывание”, исследует процесс развертывания с применением системы CI, обеспечивающей функциональную проверку.

Глава 9, “Непрерывная обратная связь”, описывает и демонстрирует использование устройств непрерывной обратной связи (например, электронной почты, RSS, X10 и Ambient Orb), способных уведомить об успехе или отказе построения.

В эпилоге описаны будущие возможности CI.

Приложения

Приложение А, “Ресурсы CI”, включает список URL, инструментов и документов о CI.

Приложение Б, “Обсуждение инструментальных средств CI”, содержит оценку различных серверов CI и соответствующих инструментальных средств, доступных на рынке. Здесь обсуждается их применимость для практик, описанных в книге, анализируются их преимущества и недостатки, а также объясняются методы использования некоторых из них.

Другие средства

В книге содержатся следующие элементы, призванные облегчить изучение изложенного материала.

- Практики. В настоящей книге рассматривается более сорока практик, относящихся к CI. Множество подзаголовков глав относятся к действиям. Рисунки в начале большинства глав иллюстрируют описываемые в ней практики и позволяют просматривать интересующие области. Например, *используйте выделенную машину для интеграционного построения* и *обновляйте код чаще* — вот два примера практик, обсуждаемых в книге.
- Примеры. На основе различных примеров на разных языках и платформах будет продемонстрировано, как применять эти практики.
- Вопросы. Каждая глава заканчивается списком вопросов, которые помогут оценить пригодность практики CI для конкретного проекта.
- Web-сайт. Web-сайт `www.integratebutton.com` предоставляет обновления, примеры кода и другие материалы для поддержки этой книги.

Что можно узнать

Читая эту книгу, можно изучить концепции и практики, позволяющие создавать слаженное, работоспособное программное обеспечение. Мы прежде всего сосредоточимся на практиках, иллюстрируя способы их применения наглядными примерами везде, где это возможно. В примерах использованы разные платформы разработки, такие как Java, Microsoft .NET и даже Ruby. В качестве основного сервера CI на протяжении всей книги используется CruiseControl (версии Java и .NET); однако мы создавали подобные примеры, используя также и другие серверы и инструменты; они перечислены на сопутствующем Web-сайте (`www.integratebutton.com`) и в приложении Б, “Обсуждение инструментальных средств CI”.

Читая эту книгу, можно научиться следующему:

- как реализовать CI, создающую *развертываемое программное обеспечение* на каждом этапе разработки;

- как CI способна *сократить время* между проявлением дефекта и обнаружением его причины, удешевляя таким образом его устранение;
- как обеспечить *качество программного обеспечения* в результате его частой интеграции, не откладывая это на последние этапы разработки.

Что не рассматривается в данной книге

Эта книга не рассматривает подробно каждый инструмент (планирование построения, среду программирования, контроль версий и т.д.), составляющий систему CI. Ее основное внимание уделено реализации практик CI, призванных выработать эффективную систему CI. Сначала обсуждаются практики CI; если некий рассматриваемый инструмент больше не используется или не соответствует текущим потребностям, просто применяйте практику, используя другой инструмент, чтобы получить тот же результат.

Кроме того, невозможно и бесполезно описывать каждый тип проверки, механизм обратной связи, автоматизированный инспектор или тип развертывания, используемый системой CI. Мы надеемся, что сосредоточившись на диапазоне ключевых практик, предоставив примеры применения методов и инструментов для интеграции баз данных, проверки, инспекции и обратной связи, мы выполнили свою основную задачу и вдохновили вас на создание приложений, столь же великолепных, как и проекты ведущих групп. Как неоднократно упоминается в книге, сопутствующий Web-сайт `www.integratebutton.com` содержит примеры, инструментальные средства и языки, которые не могут быть описаны в настоящей книге.

Авторство

Эта книга — результат совместных усилий трех авторов и одного соавтора. Я написал большинство глав. Стив Матиас занимался главами 4, 5, 7, 8 и приложением A, а также некоторыми из примеров книги. Перу Энди Гловера принадлежат главы 6, 7 и 8, он также предоставил примеры и внес вклад в другие элементы книги. Эрик Тавела написал приложение B. Таким образом, когда в предложениях используются сентенции от первого лица, следует иметь в виду конкретного автора, которому принадлежат эти слова.

Об обложке

Я был невероятно обрадован, узнав, что наша книга войдет в известную серию книг Мартина Фаулера. Поэтому я решил выбрать для обложки книги мост. Мои соавторы и я относимся к редкой породе людей, выросших в районе Вашингтона, округ Колумбия. Тем, кто вырос не в нашем округе, заметим, что эта область богата разнообразными переходами. Конкретней, мы из Северной Вирджинии, поэтому, отдавая дань этому факту, мы выбрали для обложки Природный Мост Вирджинии (Natural Bridge in Virginia). До этого я никогда не посещал его. Он имеет очень интересную историю, и мне показалось невероятным, что это полнофункциональный мост, по которому автомобили проезжают каждый день. (Безусловно, я проехал на машине по нему несколько раз.) Мне хотелось бы надеяться, что прочитав эту книгу, вы сделаете CI естественной частью вашего следующего рабочего проекта программного обеспечения.

Благодарности

Я много раз читал благодарности авторов типа “Мне никогда бы не удалось это сделать самому” и другие вещи в этом роде. Я всегда полагал, что это всего лишь ложная скромность. Но, как оказалось, я был не прав. Эта книга была титаническим свершением, которое я осуществил благодаря людям, перечисленным ниже.

Я хотел бы поблагодарить мое издательство, Addison-Wesley, и в частности выразить благодарность моему исполнительному редактору, Крису Гузикоуски (Chris Guzikowski), за его работу со мной в течение этого изнурительного процесса. Его опыт, понимание и поддержку трудно переоценить. Кроме того, мой научный редактор, Крис Зан (Chris Zahn), обеспечивал компетентные рекомендации на протяжении нескольких версий и циклов редактирования. Я также хотел бы выразить искреннюю признательность Карен Геттман (Karen Gettman), Мишель Хаусли (Michelle Housley), Джессике Д'Амико (Jessica D’Amico), Джули Нэйхил (Julie Nahil), Ребекке Гринберг (Rebecca Greenberg) и наконец, но не в последнюю очередь, моему первому исполнительному редактору Мэри О'Брайен (Mary O’Brien).

Рич Миллс (Rich Mills) предоставил для книги сервер CVS и выдавал превосходные идеи на сеансах мозгового штурма. Я также хотел бы поблагодарить моего наставника и друга Роба Дали (Rob Daly) за то, что он приобщил меня в 2002 году к профессиональному писательству, попросив составить исключительно подробные обзоры процесса программирования. Джон Стивен (John Steven) вдохновил меня на написание этой книги.

Я хотел бы выразить признательность моим соавторам, редакторам и помощникам. Со Стивом Матиасом мы провели множество бессонных ночей, создавая то, что вы читаете сегодня. Энди Гловер был нашим координатором, предоставившим свой обширный опыт разработки и проверки в процессе проектирования.

Лайза Портер, наш редактор-соавтор, неустанно совершенствовала каждый вариант книги, редактируя и предоставляя рекомендации, которые помогли улучшить ее качество. Благодарю Эрика Тавела, написавшего приложение по инструментам CI, и Левента Гарсес (Levent Gurses) за предоставленный опыт по Maven 2 в приложении B.

У нас были и электронные помощники, а также персональные технические рецензенты, поддерживавшие мгновенную обратную связь на протяжении всего проекта. Среди них Том Копланд (Tom Copeland), Роб Дали, Салли Дюваль (Sally Duvall), Каспер Хорнструп (Casper Hornstrup), Джо Хант (Joe Hunt), Айрин Джексон (Erin Jackson), Джо Кониор (Joe Konior), Рич Миллс, Лесли Пауер (Leslie Power), Дэвид Сиск (David Sisk), Карл Таллис (Carl Tallis), Эрик Тавела, Дэн Тейлор (Dan Taylor) и Сажит Васудеван (Sajit Vasudevan).

Я также хотел бы благодарить Чарльза Марри (Charles Murray) и Кристалл Белония (Cristalle Belonia) за помощь, а также Масиеж Завадски (Maciej Zawadzki) и Эрика Миник (Eric Minick) за помощь с Urbancode.

Я благодарю за поддержку многих прекрасных людей, вдохновлявших меня каждый день, а именно: Бурк Кокс (Burke Cox), Мэнди Оуенс (Mandy Owens), Дэвида Вуда (David Wood) и Рона Райта (Ron Wright). Существуют также люди, вдохновлявшие меня на протяжении многих лет, среди которых Рич Кемпбелл (Rich Campbell), Дэвид Фадо (David Fado), Майк Фрейзер (Mike Fraser), Брент Гендлеман (Brent Gendleman), Джон Хьюз (Jon Hughes), Джеф Хванг (Jeff Hwang), Черри Хванг (Sherry Hwang), Сэнди Кайли (Sandi Kyle), Брайан Лионс (Brian Lyons), Сьюзен Мейсон (Susan Mason), Брайан Мессер (Brian Messer), Сэнди Миллер (Sandy Miller), Джон Ньюман (John Newman), Маркус Оуэн (Marcus Owen), Крис Паинтер (Chris Painter), Полетт Роджерс (Paulette Rogers), Марк Симоник (Mark Simonik), Джо Стасник (Joe Stusnick) и Майк Траил (Mike Trail).

Я также ценю помощь группы технических рецензентов издательства Addison-Wesley, включая Скотта В. Эмблера (Scott Ambler), Бреда Апплетона (Brad Appleton), Джона Эвеса (Jon Eaves), Мартина Фаулера, Пола Холсера (Paul Holser), Пола Джулиуса (Paul Julius), Кирка Кноршильда (Kirk Knoernschild), Майка Мелия (Mike Melia), Джулиан Симпсон (Julian Simpson), Энди Тригг (Andy Trigg), База Водд (Bas Vodde), Майкла Вард (Michael Ward) и Джейсон Уип (Jason Yip).

Я хочу поблагодарить посетителей конференции CITCON, Чикаго 2006 год, за обмен опытом в области CI и проверки. В частности, я очень признателен Полу Джулиусу и Джеффри Фредерику (Jeffrey Frederick) за организацию конференции и всем остальным, участвовавшим в этом событии.

И наконец, я хотел бы поблагодарить Джейн за ее постоянную поддержку и помощь в преодолении трудностей на протяжении создания этой книги.

Поль М. Дюваль, Ферфакс, Вирджиния
Март 2007

Соглашения, принятые в этой книге

При оформлении книги использованы соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется *курсив*.
- Текст программ, функций, переменных, URL, Web-страниц и другой код представлен моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** шрифтом.
- Знакоместо в описаниях синтаксиса выделено *курсивом*. Это указывает на необходимость заменить его фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте `BINDSIZE=(`*`максимальная ширина колонки`*`)*(`*`номер колонки`*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).
- Разрыв слишком длинных строк кода, не помещающихся на странице, обозначен специальным символом ↳.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что бы еще вы хотели увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш Web-сервер, оставив на нем свои замечания, — одним словом, любым удобным для вас способом дайте нам знать, нравится вам эта книга или нет, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

Часть I

Основы CI – принципы и практики

Глава 1

Первые шаги

Сначала овладейте основными принципами.

Лэрри Бирд (Larry Bird) (Профессиональный американский баскетболист)

Кэйси Сиерра (Kathy Sierra), основатель `javaranch.com`, пишет в своем блоге: "Существует большая разница между сказать «Ешьте по одному яблоку в день» и фактическим поеданием яблок."[1] То же можно сказать и о фундаментальных практиках проекта программного обеспечения. Нередко можно услышать, что "проверки неэффективны" или "обзоры кода — пустая трата времени", или что частое построение программного обеспечения — плохая практика. Однако фундаментальные методологии, по-видимому, следует использовать на практике, а не проповедовать их, поскольку частота их применения в проектах довольно низка.

Если необходимо выполнять частые интеграционные построения так, чтобы они перестали быть *событием* в проекте (включая компиляцию, перестройку базы данных, автоматизированные проверки и инспекции, развертывание программного обеспечения и получение обратной связи), то *непрерывная интеграция* (Continuous Integration — CI) может в этом помочь. В данной главе продемонстрированы общие средства, доступные в системах CI, основанных на фундаментальных практиках построения программного обеспечения.

Понять основные принципы CI довольно просто, к тому же введение фундаментальных практик разработки программного обеспечения в проект не займет много времени.

Стройте программное обеспечение при каждом изменении

При чтении книг мне нравится сначала рассмотреть пример, а затем изучать все вопросы, лежащие в его основе, поскольку, на мой взгляд, пример обеспечивает контекст для них. Автор описывает сценарий CI на основании типичной реализации, а поскольку существуют различные способы реализации системы CI, это должно дать общее представление о составе типичной системы.

[1] Источник: *`http://headrush.typepad.com/`*.

Что такое *построение*?

Построение (build) — это намного больше, чем *компиляция* (compile) (или разнообразные лингвистические вариации). Построение, кроме всего прочего, может состоять из компиляции, проверки, инспекции и развертывания. Построение — это скорее процесс сборки исходного кода в единый модуль и проверки программного обеспечения на способность к совместной работе.

Сценарий CI начинается с передачи разработчиком исходного кода в хранилище. В типичном проекте люди, исполняющие многие роли в них, могут вносить изменения, которые приводят к запуску цикла CI: разработчики изменяют исходный код; администраторы базы данных (DBA) — определения таблиц; группы построения и развертывания — файлы конфигурации, группы взаимодействия — спецификации DTD/XSD и т.д.

Поддержка актуальности примеров

Риск приведения "практического" примера в книге, особенно по такой динамичной теме, как CI, заключается в том, что он может быстро устареть. Чтобы актуализировать материалы, в которые могут быть внесены изменения после выхода настоящей книги, мы размещаем их на сопутствующем Web-сайте книги, `www.integratebutton.com`, включая не только примеры, CruiseControl и Ant, но и многие другие серверы и инструменты CI.

Сценарий CI, как правило, имеет следующие этапы.

1. Сначала разработчик передает код в хранилище с контролем версий. Тем временем сервер CI на машине интеграционного построения опрашивает это хранилище на предмет изменений (например, каждые несколько минут).
2. Вскоре после передачи кода сервер CI обнаруживает изменения в хранилище с контролем версий. Сервер CI получает последнюю копию кода из хранилища и запускает сценарий построения, который интегрирует программное обеспечение.
3. Сервер CI отрабатывает обратную связь, посылая по электронной почте результаты построения соответствующим участникам проекта.
4. Сервер CI продолжает опрашивать хранилище с контролем версий на предмет новых изменений.

Элементы системы CI иллюстрирует рис. 1.1.

В следующих разделах инструменты и элементы, представленные на рис. 1.1, описываются более подробно.

Разработчик

Как только разработчик внесет все изменения, связанные с поставленной задачей, он выполняет закрытое построение (которое интегрирует изменения с остальной частью кода группы), после чего передает измененный код в хранилище с контролем версий. Данный шаг может быть предпринят в любой момент, поскольку он не влияет на последующие этапы процесса CI. Интеграционное построение не происходит без наличия изменений, внесенных в хранилище с контролем версий.

Листинг 1.1 демонстрирует пример выполнения закрытого построения при помощи вызова сценария построения Ant из командной строки. Обратите внимание, что этот сценарий получает последние обновления из хранилища с контролем версий Subversion.

Рис. 1.1. Компоненты системы CI

Листинг 1.1. Запуск закрытого построения с использованием сценария Ant

```
> ant integrate
Buildfile: build.xml
clean:
svn-update:
all:
compile-src:
compile-tests:
integrate-database:
run-tests:
run-inspections:
package:
deploy:
BUILD SUCCESSFUL
Total time: 3 minutes 13 seconds
```

Раннее обнаружение проблем при частом построении

Как только вы автоматизируете свой процесс построения, чтобы он мог запускаться одной командой, все готово к переходу на CI. Выполняя автоматизированное построение при внесении любых изменений в проект, группа получит от системы управления версиями ответы на следующие вопросы.

- Все ли компоненты программного обеспечения способны работать вместе?
- В чем сложности используемого кода?
- Жестко ли группа придерживается установленных стандартов программирования?
- Насколько код охвачен автоматизированными проверками?
- Действительно ли все проверки оказались успешными после последнего изменения?

- Удовлетворяет ли приложение всем требованиям производительности?
- Остались ли какие-то проблемы со времени последнего развертывания?

Знание того, что программное обеспечение было успешно "построено" с учетом последних изменений, крайне ценно, но знание того, что программное обеспечение было построено *правильно*, просто неоценимо, поскольку накопление дефектов в базовом коде программного обеспечения допускается лишь до некоторой степени. Причина, по которой столь выгодно *непрерывное* построение, заключается в быстрой обратной связи, позволяющей находить и устранять проблемы в течение цикла разработки.

После успешного выполнения закрытого построения вы можете передавать в хранилище новые и измененные файлы. Большинство систем контроля версий поддерживают простые команды, позволяющие запускать эти процессы, как показано в листинге 1.2, где используется Subversion.

Листинг 1.2. Передача изменений в хранилище Subversion

```
> svn commit –m "Added CRUD capabilities to DAO"
Sending src\BeerDaoImpl.jaca
Transmitting file data.

Committed revision 52.
```

Вы можете также запускать свой сценарий построения и передавать изменения в хранилище, используя вашу *интегрированную среду разработки* (Integrated Development Environment — IDE). Только удостоверьтесь сначала, что можете выполнить оба действия из командной строки, чтобы не иметь жесткой зависимости от IDE или системы контроля версий.

Хранилище с контролем версий

Не забывайте, что для непрерывной интеграции необходимо использовать хранилище с контролем версий. Фактически, даже если вы не применяете CI, хранилище с контролем версий, как правило, используется в проекте. Его задача заключается в контроле изменений исходного кода и других элементов программного обеспечения (например, документации), а также в управлении доступом. Это предоставляет "единый источник" для всего исходного кода, где он будет доступен из любого места. Хранилище с контролем версий позволяет возвращаться к прежним версиям исходного кода и других файлов.

Вы запускаете CI как основную линию хранилища с контролем версий (например, Head/Trunk в таких системах, как CVS и Subversion). Существуют различные типы систем контроля версий, которые вы также можете использовать. Мы применяем систему Subversion для большинства примеров в книге из-за ее набора возможностей и бесплатного доступа. К другим инструментам *управления конфигурацией программного обеспечения* (Software Configuration Management — SCM) и контроля версий относятся CVS, Perforce, PVCS, ClearCase, MKS и Visual SourceSafe. Эффективные методы управления конфигурацией программного обеспечения можно изучать по книге Стивена Беркзука (Stephen Berczuk) и Бреда Апплетона (Brad Appleton) *Software Configuration Management Patterns*.

Сервер CI

Сервер CI выполняет интеграционное построение всякий раз, когда в хранилище с контролем версий передаются изменения. Как правило, вы настраиваете сервер CI, чтобы проверять изменения в хранилище с контролем версий каждые несколько минут или

около того. Сервер CI получает файлы исходного кода и запускает сценарий или сценарии построения. Серверы CI допускают планирование, обеспечивая построение с обычной частотой, например каждый час (однако обратите внимание, это не непрерывная интеграция). Кроме того, серверы CI обычно предоставляют удобную панель, где отображаются результаты построения. Хотя это и рекомендуется, сервер CI не обязан выполнять непрерывную интеграцию. Но вы можете писать свои собственные специальные сценарии и, кроме того, запускать вручную интеграционное построение всякий раз, когда в хранилище происходит изменение. Использование сервера CI уменьшает количество специальных сценариев, которые в противном случае вам пришлось бы написать. Доступно множество серверов CI, включая бесплатные с открытым исходным кодом. Листинг 1.3 демонстрирует пример использования файла `config.xml` на сервере CruiseControl для опроса хранилища Subversion на предмет изменений.

Листинг 1.3. Файл `config.xml` сервера CruiseControl для опроса хранилища Subversion

```
<project name="brewery" >
    <listeners>
        <currentbuildstatuslistener
            file="logs/${project.name}/status.txt"/>
    </listeners>
    <modificationset quietperiod="30">
        <svn RepositoryLocation="http://build.ib.com/trunk/brewery"
            username="bfranklin"
            password="G0Fly@Kite"/>
    </modificationset>
    <schedule interval="300">
        <ant anthome="apache-ant-1.6.5"
            buildfile="bld-{project.name}.xml"/>
    </schedule>
    <log dir="logs/${project.name}">
        <merge dir="projects/${project.name}/impl/logs/junit"/>
        <merge dir="projects/${project.name}/impl/logs/cobertura"/>
    </log>
    <publishers>
        <artifactspublisher dir="projects/${project.name}/impl/logs"
            dest="artifacts/${project.name}"/>
        <artifactspublisher dir="projects/${project.name}/impl/logs"
            dest="artifacts/${project.name}"/>
    </publishers>
</project>
```

В листинге 1.3 атрибут `interval` задачи `schedule` указывает, как часто сервер CruiseControl будет проверять изменения в хранилище Subversion (в этом примере через 300 секунд). Если сервер CruiseControl находит любые модификации, он осуществляет *делегирующее построение* (delegating build), вызываемое при помощи атрибута `buildfile` в коде листинга 1.3. Делегирующее построение (не показанное на схеме) получает последний исходный код из хранилища и запускает файл `build` проекта, подобный представленному в листинге 1.3. Для администрирования другие серверы CI могут использовать Web-ориентированный интерфейс конфигурации или другой интерфейс. Сервер CruiseControl

поставляется с Web-приложением, что позволяет просматривать результаты последнего построения и отчеты (например, отчеты проверок и инспекции). Рис. 1.2 иллюстрирует пример результатов построения проекта на сервере CruiseControl.

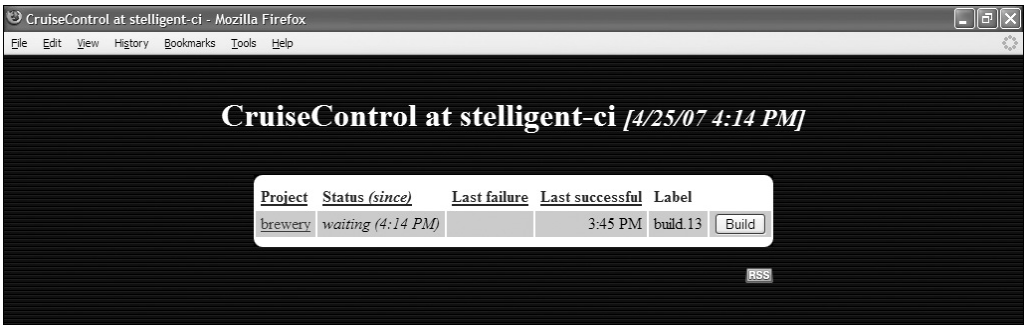


Рис. 1.2. Панель CruiseControl, отображающая состояние последнего построения

Сценарий построения

Сценарий построения (build script) — это единый сценарий, или набор сценариев, используемый для компиляции, проверки инспекции и развертывания программного обеспечения. Вы можете применять сценарий построения без реализации системы. К инструментам, позволяющим автоматизировать цикл построения программного обеспечения, относятся Ant, NAnt, make, MSBuild и Rake, но сами по себе они не обеспечивают CI. Для построения программного обеспечения некоторые используют IDE, однако с тех пор как процесс CI перестал быть "ручным" исключительно с применением средств построения, обладающих IDE, не будем сбрасывать его со счетов. Если быть честным, то используя IDE для построения, вы можете сделать то же самое, что и *без его применения*. Листинг 1.4 демонстрирует пример оболочки сценария Ant, который запускает процессы, обычно выполняемые в ходе закрытого построения[2].

Листинг 1.4. Оболочка сценария Ant, осуществляющего построение

```
<?xml version="1.0" encoding="iso-8859-1"?>
<project name="brewery" default="all" basedir=".">
    <target name="clean" />
    <target name="svn-update" />
    <target name="all" depends="clean,svn-update"/>
    <target name="compile-src" />
    <target name="compile-tests" />
    <target name="integrate-database" />
    <target name="run-tests" />
    <target name="run-inspections" />
    <target name="package" />
    <target name="deploy" />
</project>
```

[2] Более подробный пример предоставлен по адресу `www.integratebutton.com`.

Механизм обратной связи

Одной из ключевых задач CI является обратная связь при интеграции, поскольку вы, естественно, хотите как можно скорее узнать, не было ли проблем с последним построением. Оперативное получение такой информации позволяет быстро устранить проблему. Рис. 1.3 иллюстрирует механизм обратной связи по электронной почте. Большее количество устройств обратной связи демонстрируется в главе 9, “Непрерывная обратная связь”. К другим механизмам обратной связи относится *служба коротких сообщений* (Short Message Service — SMS) и RSS (Really Simple Syndication).

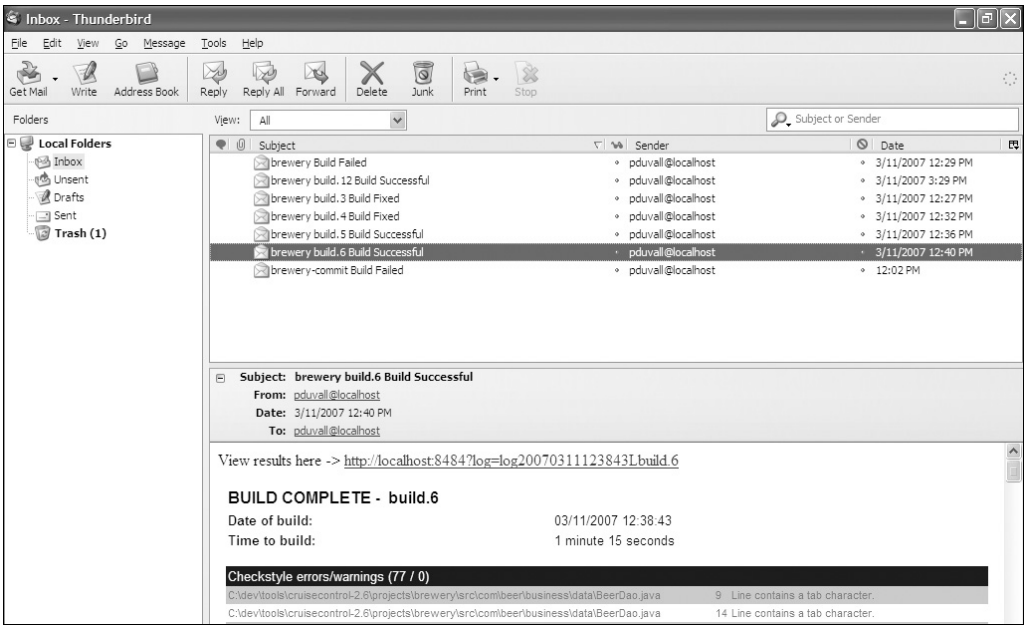


Рис. 1.3. Сообщение электронной почты, полученное от сервера CI

Листинг 1.5 содержит пример использования сервера CI CruiseControl для передачи сообщения электронной почты участникам проекта.

Листинг 1.5. Файл `config.xml` сервера CruiseControl для настройки передачи электронной почты

```
<project>
...
    <publishers>
        <htmlemail
            css="./webapps/cruisecontrol/css/cruisecontrol.css"
            mailhost="localhost"
            xsldir="./webapps/cruisecontrol/xsl"
            returnaddress="pduvall@localhost"
            buildresultsurl="http://localhost:8080"
            mailport="225"
            username="pduvall"
            password="password"
```

```
            reportsuccess="always"
            spamwhilebroken="true">
            <always address="pduvall@localhost"/>
            <always address="aglover@localhost"/>
        </htmlemail>
    </publishers>
...
</project>
```

Машина интеграционного построения

Машина интеграционного построения (integration build machine) — это отдельная машина, единственной задачей которой является интеграция программного обеспечения. Машина интеграционного построения содержит сервер CI, который опрашивает хранилище с контролем версий.

Средства интеграционного построения

Теперь, рассмотрев пример построения, мы можем погрузиться в средства CI. Существует только четыре компонента, *обязательных* для CI.

- Подключение к хранилищу с контролем версий.
- Сценарий построения.
- Некий вид механизма обратной связи (типа электронной почты).
- Процесс интеграции изменений исходного кода (ручной или сервер CI).

Это "опорное" (bare-bone) поведение — ключ к эффективной системе CI. Если при каждом изменении ваша система контроля версий запускает автоматизированное построение, то вы можете добавить в систему CI и другие средства.

Выполняя автоматизированную и непрерывную интеграцию базы данных, проверку, инспекцию, развертывание и обратную связь, ваша система CI способна уменьшить общие риски в проекте, повышая таким образом эффективность разработки и коммуникабельность разработчиков. Некоторые компоненты зависят от других составляющих; например, автоматизированная проверка зависит от компиляции исходного кода.

Такой повторяющийся процесс может снизить риски в течение цикла разработки. Процессы, происходящие внутри цикла, подробно описаны далее.

Компиляция исходного кода

Непрерывная компиляция исходного кода — одно из наиболее простых и обычных средств систем CI. Фактически это настолько обычно, что практически стало синонимом CI. Компиляция подразумевает создание исполняемого кода из исходного кода, читаемого человеком. Хотя непрерывная интеграция — это намного больше, чем компиляция исходного кода, в результате быстрого увеличения количества применяемых *динамических* (dynamic) языков (Python, PHP, Ruby и т.д.) компиляция в этих системах несколько отличается. Хотя с использованием динамических языков вы не создаете бинарный код, многие из них позволяют выполнять строгую проверку, которую в контексте таких языков можно рассматривать как компиляцию. Несмотря на эту тонкость, динамические

языковые системы обеспечивают преимущества в других действиях, выполненных в течение построения.

Кнопка <Integrate>

Кнопка <Integrate> (рис. 1.4) — это отображение полнофункционального и автоматизированного интеграционного построения, позволяющая превратить интеграцию в привычную операцию. Сюда относится большинство процессов, гарантирующих, что ваше программное обеспечение будет работать должным образом. Вы можете компилировать, перестраивать базу данных с проверочными данными, выполнять проверки, инспекцию, развертывание и поддерживать обратную связь. Автоматизируя построение, вы можете запускать большинство этих процессов, нажав *одну кнопку*.

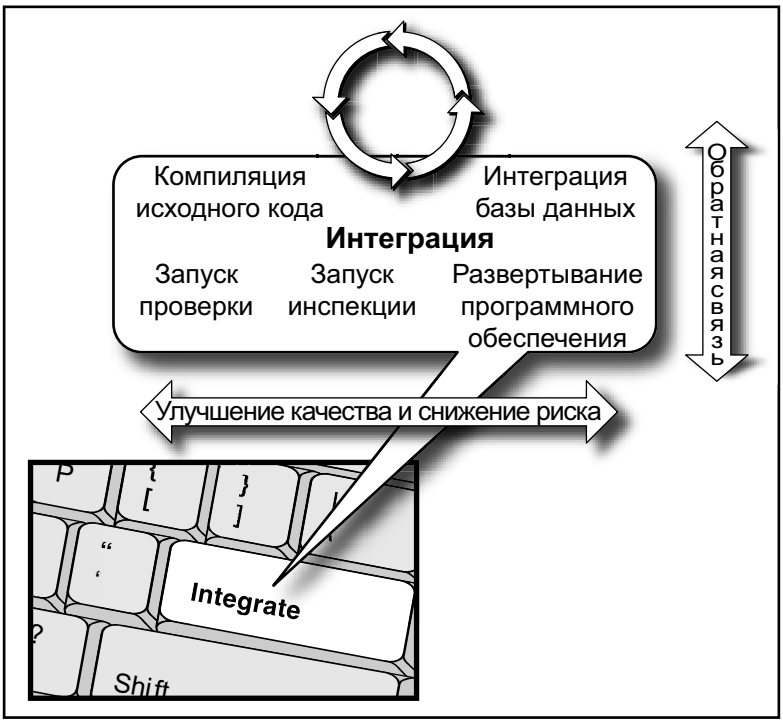


Рис. 1.4. Отображение кнопки <Integrate>

Интеграция базы данных

Некоторые считают интеграцию исходного кода и интеграцию базы данных совершенно разными процессами, *всегда* выполняемыми разными группами. Но это не самый удачный подход, поскольку база данных (если вы используете ее в своем проекте) — неотъемлемая часть прикладного программного обеспечения. Используя систему CI, вы можете гарантировать интеграцию базы данных при помощи единого источника: вашего хранилища с контролем версий.

Рис. 1.5 демонстрирует обеспечение системой CI непрерывной интеграции базы данных в процессе построения. Мы выполняем исходный код базы данных — сценарии *языка определения данных* (Data Definition Language — DDL), сценарии *языка обработки данных* (Data Manipulation Language — DML), определение хранимых процедур, — выполняя разделение и т.д. точно так же, как и любой другой исходный код в системе. Например, когда участник проекта (разработчик или администратор базы данных) изменяет сценарий базы данных и передает его системе контроля версий, тот же самый сценарий построения, который интегрирует исходный код, перестроит базу данных и данные в ходе процесса интеграционного построения.

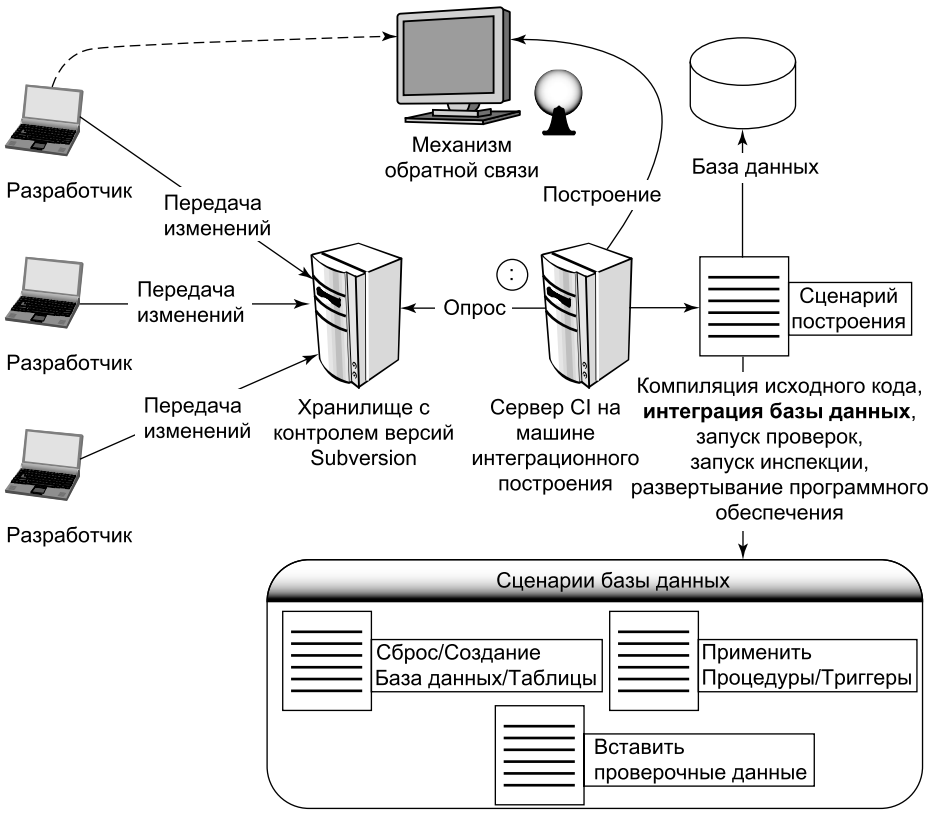


Рис. 1.5. Проект интеграции базы данных

Листинг 1.6 демонстрирует, как сбросить и построить базу данных MySQL, используя задачу Ant `sql`. Это намного больше, чем вы будете делать для перестройки баз данных и их проверки. Данный статический пример кода приведен исключительно в демонстрационных целях.

Листинг 1.6. MySQL и Ant

```
<target name="db:create-database">
    <sql driver="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/"
        userid="root"
```

```
        password="sa"
        classpathref="db.lib.path"
        delimiter=";">
        <fileset file="${database.dir}/drop-database.sql"/>
        <fileset file="${database.dir}/create-database.sql "/>
    </sql>
</target>
```

Более подробные примеры, подходы и преимущества интеграции базы данных рассматриваются в главе 5, “Непрерывная интеграция баз данных”.

Проверка

Многие считают, что CI возможна *без* автоматизированной непрерывной проверки. Но мы не можем согласиться с этим. Без автоматизированной проверки разработчикам и другим участникам проекта трудно гарантировать корректность изменений программного обеспечения. Большинство разработчиков в проектах с использованием системы CI применяют для проверки модулей такие инструменты, как JUnit, NUnit и другие типа *x*Unit. Кроме того, из системы CI можно запускать проверки различных категорий, чтобы ускорить построение. Данные категории могут включать модули, компоненты, систему, нагрузку и эффективность, защиту и другие. Более подробно большинство этих проверок обсуждается в главе 6, “Непрерывная проверка”. Рис. 1.6 демонстрируют пример отчета JUnit, который сервер CI, такой как CruiseControl, может формировать в ходе интеграционного построения.

Рис. 1.6. Отчет о проверке модуля с использованием JUnit

Листинг 1.7 демонстрирует пример кода, который запускает пакет проверки JUnit и создает отчет, приведенный на рис. 1.6, с использованием задачи Ant.

Листинг 1.7. Ant и JUnit

```
<?xml version="1.0" encoding="iso-8859-1"?>
    <target name="run-tests">
        <mkdir dir="${logs.junit.dir}" />
        <junit fork="yes" haltonfailure="true" dir="${basedir}"
                    printsummary="yes">
            <classpath refid="test.class.path" />
            <classpath refid="project.class.path" />
```

```
            <formatter type="plain" usefile="true" />
            <formatter type="xml" usefile="true" />
            <batchtest fork="yes" todir="${logs.junit.dir}">
                <fileset dir="${test.unit.dir}">
                    <patternset refid="test.sources.pattern" />
                </fileset>
            </batchtest>
        </junit>
        <mkdir dir="${reports.junit.dir}" />
        <junitreport todir="${reports.junit.dir}">
            <fileset dir="${logs.junit.dir}">
                <include name="TEST-*.xml" />
                <include name="TEST-*.txt" />
            </fileset>
            <report format="frames" todir="${reports.junit.dir}" />
        </junitreport>
    </target>
</project>
```

Инспекция

Автоматизированная инспекция кода (например, статический и динамический анализ) применяется для улучшения качества программного обеспечения за счет применения правил. Например, в проекте могло бы существовать правило, согласно которому размер класса не должен превышать 300 строк некомментированного кода. Вы можете использовать свою систему CI для автоматической инспекции базового кода на предмет соответствия этим правилам. Более подробно инструменты и методы инспекции обсуждаются в главе 7, “Непрерывная инспекция”.

Типичный отчет об инспекции программного обеспечения, представленный на рис. 1.7, был создан системой Checkstyle по результатам инспекции кода Java. Использование отчетов, подобных этому, может обеспечить непрерывный мониторинг кода на предмет соответствия стандартам и качественным показателям.

Листинг 1.8 демонстрирует пример применения инструмента статического анализа кода Checkstyle совместно с Ant. Этот код создает отчет, приведенный на рис. 1.7.

Листинг 1.8. Пример применения Checkstyle с использованием Ant[3]

```
<target name="run-inspections">
    <taskdef resource="checkstyletask.properties"
                      classpath="${checkstyle.jar}"/>
    <checkstyle config="${basedir}/checkstyle-rules.xml"
                       failOnViolation="false">
        <formatter toFile="${checkstyle.data.file}" type="xml" />
        <fileset dir="${src.dir}" includes="**/*.java" />
    </checkstyle>
    <xslt taskname="checkstyle"
        in="${checkstyle.data.file}"
```

[3] Из книги Поля Дюваля *Automation for the People: Continuous Inspection*. Издательство IBM developerWorks, август 2006 года, доступно на http://www-128.ibm.com/developerworks/java/library/j-ap08016/.

```
        out="${checkstyle.report.file}"
        style="${checkstyle.xsl.file}" />
</target>
```

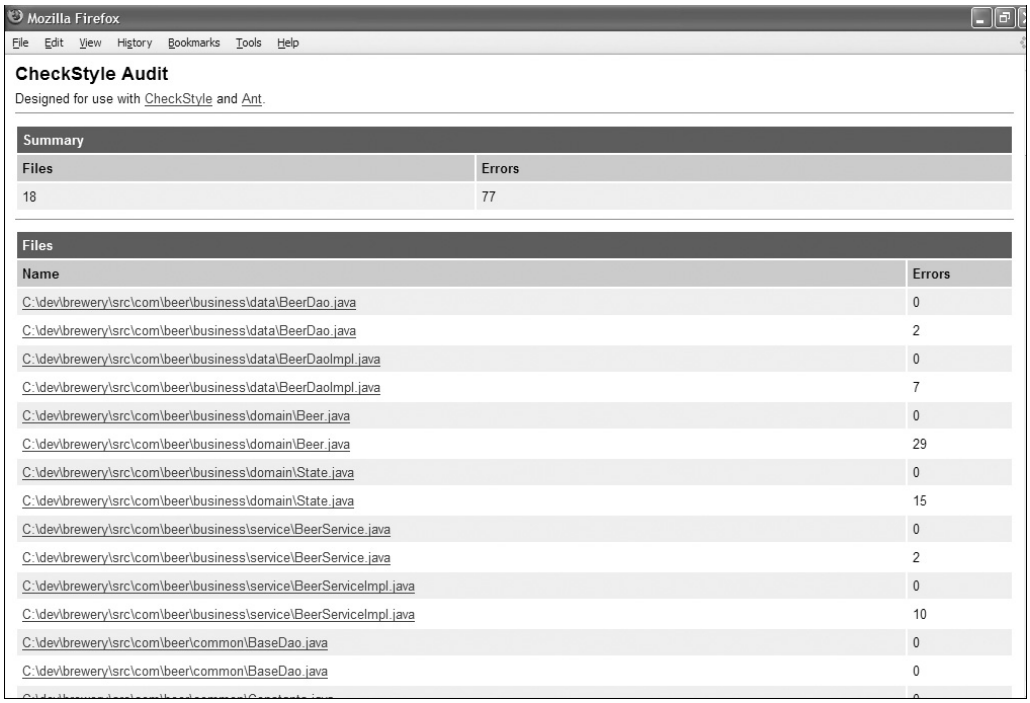


Рис. 1.7. Отчет автоматизированной инспекции с использованием инструмента Checkstyle

Развертывание

Большинство процессов подразумевает нечто, известное как *развертывание* (deployment). Фактически многие из процессов, обсуждаемых в этом разделе, являются частью процесса развертывания. Непрерывное развертывание позволяет вам предоставлять работоспособное развертываемое программное обеспечение в любой момент. Это означает, что ключевая цель системы CI заключается в том, чтобы создать связные артефакты программного обеспечения с учетом последних изменений и делать их доступными для системы проверки.

Перед развертыванием следует выполнить множество действий: проверить файлы исходного кода в хранилище с контролем версий, осуществить построение, добиться успешного выполнения всех проверок и инспекций, а также пометить релиз и организовать развертываемые файлы.

Система CI вполне может автоматически произвести развертывание (или установку) файлов в соответствующей среде, как демонстрирует рис. 1.8. Кроме того, развертывание должно подразумевать возможность автоматической отмены всех изменений (откат), сделанных в его процессе. Обратите внимание, что вы можете использовать несколько разных операционных сред при разработке (например, Jetty, как проиллюстрировано на рис. 1.8),

интеграции и проверке (Tomcat). Независимо от того, автоматизировано ли само построение и каковы его параметры, осуществляется оно в подобных средах. Более подробная информация об этих стратегиях приведена в главе 8, “Непрерывное развертывание”.

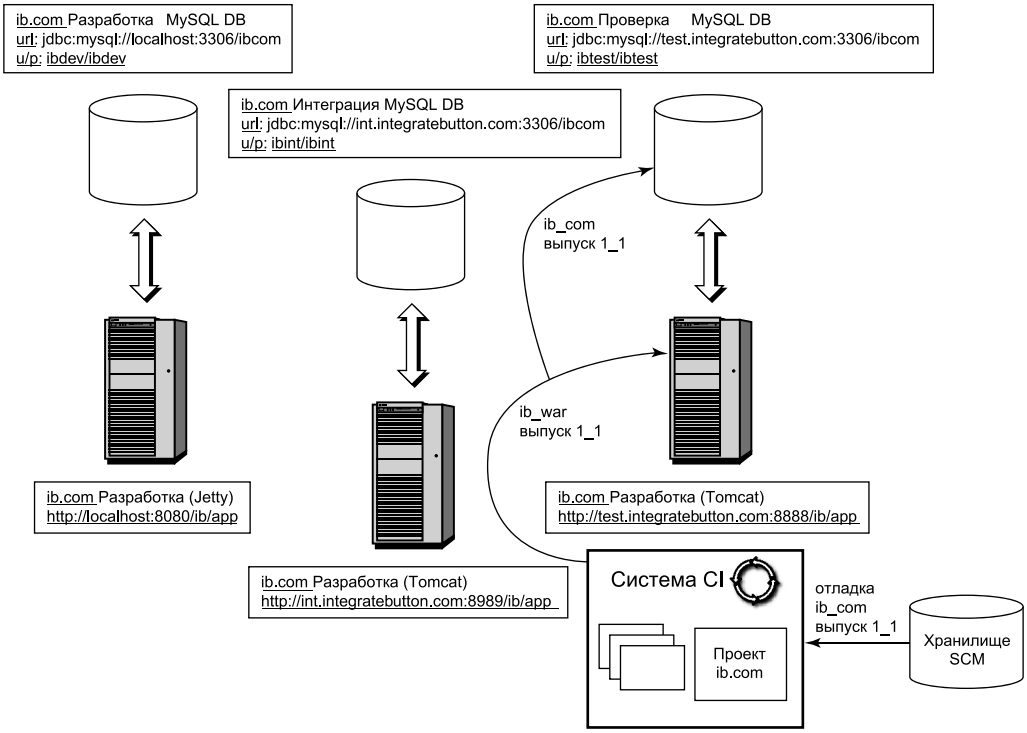


Рис. 1.8. Среда развертывания

Листинг 1.9 демонстрирует применение инструмента по имени Cargo, который обеспечивает взаимодействие между Ant и Web-контейнером. В данном случае осуществляется развертывание на сервере Tomcat. Инструмент Cargo обеспечивает взаимодействие с большинством популярных Web-контейнеров.

Листинг 1.9. Развертывание на сервере Tomcat с использованием Ant и Cargo

```
<target name="deploy">
    <cargo containerId="tomcat5x" action="start"
        wait="false" id="${tomcat-refid}">
        <zipurlinstaller installurl="${tomcat-installer-url}"/>
        <configuration type="standalone" home="${tomcatdir}">
            <property name="cargo.remote.username" value="admin"/>
            <property name="cargo.remote.password" value=""/>
            <deployable type="war" file="${wardir}/${warfile}"/>
    </configuration>
    </cargo>
</target>
```

Документирование и обратная связь

Большинство разработчиков твердо убеждены, что наилучшая документация в исходном коде — это простой и краткий код с хорошо проработанными именами классов, переменных и методов. Система CI может обеспечивать возможности документирования безо всяких проблем. Для создания документации вы можете использовать такие инструменты, как Maven, Javadoc или NDoc. Кроме того, существуют средства, позволяющие создавать схемы классов и другую информацию, причем на основании переданного им исходного кода из хранилища с контролем версий. Вам понравится возможность получения документации практически в реальном масштабе времени из исходного кода и состояния проекта при помощи системы CI. Вы можете указать, чтобы артефакты вашей документации создавались периодически, а не непрерывно.

Критически важной способностью хорошей системы CI является *скорость*. Смысл системы CI заключается в обеспечении своевременной обратной связи для разработчиков и остальных участников проекта. Систему CI довольно просто нагрузить так, что для завершения цикла понадобится слишком много времени (особенно под конец проекта), поэтому необходим разумный баланс между глубиной процесса CI и его быстродействием. Это особенно важно при непрерывной проверке. Более подробная информация о методах быстрого построения приведена в главах 4, “Построение программного обеспечения при каждом изменении”, и 6, “Непрерывная проверка”.

Резюме

Эта глава содержит краткий обзор средств непрерывной интеграции. Здесь также описано, как включить в систему CI дополнительные процессы, например интеграцию базы данных, проверку, инспекцию, развертывание и обратную связь. В остальной части книги исследуются детали каждого из этих процессов, связанных с разработкой программного обеспечения с использованием CI.

Вопросы

Откуда вы знаете, что осуществляете непрерывную интеграцию правильно? Приведенные ниже вопросы помогут выяснить, что отсутствует в ваших проектах.

- Используете ли вы хранилище с контролем версий (или инструмент SCM)?
- Автоматизирован ли и цикличен ли процесс построения вашего проекта? Осуществляется ли он полностью без вашего вмешательства?
- Пишете и запускаете ли вы автоматизированные проверки?
- Является ли запуск проверок частью вашего процесса построения?
- Используете ли вы стандарты программирования и разработки?
- Автоматизирован ли ваш механизм обратной связи?
- Используете ли вы отдельную машину для интеграционного построения программного обеспечения?

Глава 2

Введение в непрерывную интеграцию

Предположение — источник всех ошибок.

Закон Ветерна для отложенного правосудия

На заре своей карьеры я узнал, что создание хорошего программного обеспечения сводится к неукоснительному соблюдению фундаментальных практик, причем *независимо от специфической технологии*. Исходя из моего опыта, одной из страшнейших проблем разработки программного обеспечения является *предположение* (assuming). Если вы предполагаете, что методу в качестве параметра будет передано правильное значение, то с этим методом будут проблемы. Предположите, что разработчики при программировании будут следовать стандартам, и получите проблемы при поддержке программного обеспечения. Предположите, что файлы конфигурации не изменились, и вы потратите напрасно драгоценные часы, выслеживая причину проблемы, которой не существует. Делая предположения в разработке программного обеспечения, мы серьезно рискуем потратить время и силы впустую.

Ограничение предположений

Перестраивая программное обеспечение *после каждого изменения* в системе контроля версий, непрерывная интеграция способна уменьшать количество предположений в проекте.

Мы можем полагать, что последняя, самая лучшая технология будет "серебряной пулей", способной решить все наши проблемы, но это окажется не так. В одной из компаний моей главной обязанностью, в частности, был поиск и внедрение передовых практик разработки программного обеспечения. Через некоторое время мы уже были способны реализовать в проектах многие практики, широко применяемые для разработки хорошего ПО. Работая над многими разными проектами, в которых использовались различные методологии, я обнаружил, что итерационные проекты, где применялся *рациональный унифицированный процесс* (Rational Unified Process — RUP) и *экстремальное программирование* (eXtreme Programming — XP), как правило, осуществлялись лучше всего (в наших случаях), поскольку все риски при этом оказывались снижены. Разработка программного обеспечения требует планирования изменений, непрерывного наблюдения за результатами и инкрементного (порциями) исправления кода на основании последних. Именно так работает CI. Непрерывная интеграция — это воплощение тактики, которая позволяет разработчикам программного обеспечения, вносить изменения в код, причем разделять его на небольшие фрагменты, чтобы гарантировать *немедленную обратную связь*, обеспечивающую оперативное внесение корректирующих изменений.

Непрерывная интеграция — это фундаментальная концепция. Возможно, и не самая изящная среди способов разработки программного обеспечения, но в нынешних сложных проектах жизненно важная. Нередко *пользователи* программного обеспечения говорят мне: "Ого! Мне понравился ваш способ интеграции программного обеспечения в последнем выпуске". Но когда этого не происходит, может показаться, что все усилия за кулисами не заслуживают внимания. Однако любой, кто *разрабатывал* программное обеспечение, используя такую практику, как CI, восхищался однозначным и воспроизводимым процессом построения, запускаемым при изменении содержимого в хранилище с контролем версий.

Непрерывная интеграция как основной залог качества

Некоторые рассматривают CI как процесс объединения компонентов программного обеспечения. Однако по нашему мнению, CI представляет собой основу методики разработки, поскольку гарантирует высокое качество программного обеспечения за счет выполнения построения при каждом изменении. Определить качество программного обеспечения может быть столь же просто, как и проверить результат последнего интеграционного построения.

Потратив *некоторое* время на малопривлекательные фундаментальные концепции средств разработки программного обеспечения, можно уделить *больше* времени тем действиям, которые делают нашу работу интересной и привлекательной. Но если мы не уделим внимания основным принципам, таким как определение среды разработки и построение программного обеспечения, мы будем вынуждены впоследствии выполнять низкоуровневые задачи, и, как правило, в самое неподходящее время (например, непосредственно перед сдачей программного обеспечения заказчику). Именно в этот момент и происходят ошибки. Дисциплина, соблюдаемая при окончательном построении, освободит вас от беспокойства по поводу того, сохранило ли программное обеспечение работоспособность. Это

похоже на тренировку — нужна самодисциплина, много тяжелого труда, но вы будете в форме, когда придет время большой игры.

В этой главе мы попытаемся ответить на вопросы, которые могут возникнуть при реализации в проекте решения о применении практики CI. Здесь приводится краткий обзор преимуществ и недостатков CI, а также рассматриваются способы ее дополнения другими практиками разработки программного обеспечения. CI — это не практика, которая может быть внедрена в проект "мастером построения" и относительно забыта. Она повлияет на каждого участника группы разработки программного обеспечения, поэтому мы рассматриваем CI с учетом того, что ее реализацией должны заняться все участники группы.

Что представляет собой типичный рабочий день с использованием CI? Давайте воспользуемся для этого опытом Тима.

День из жизни CI

Открыв дверь своего кабинета в компании, Тим сразу видит широкий монитор, отображающий информацию о его проекте в реальном времени. На мониторе видно, что последнее интеграционное построение было успешно выполнено на сервере CI несколько минут назад. Представлен также список последних качественных показателей, включая приверженность стандартам проектирования и программирования, дублирование кода и т.д. Тим — один из 15 разработчиков проекта на языке Java, создающий программное обеспечение для управления пивоваренным заводом в оперативном режиме. Некоторые из действий Тима на протяжении его рабочего дня представлены на рис. 2.1.

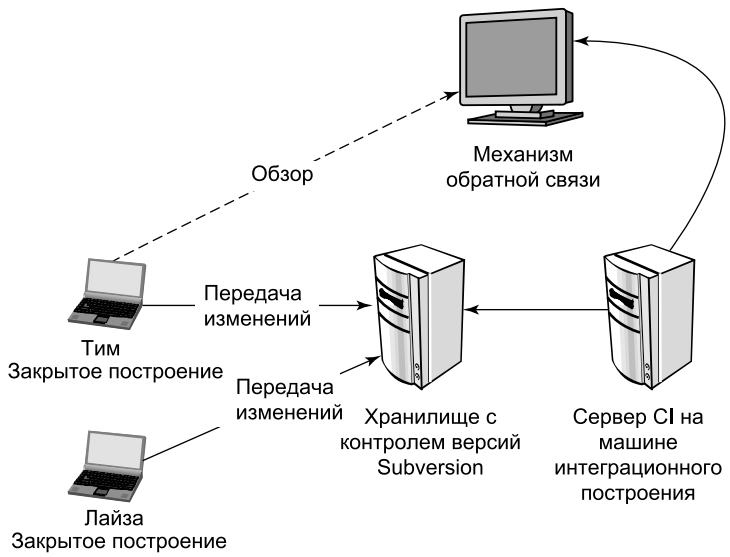


Рис. 2.1. Один день из жизни

Начиная свой день, Тим проводит рефакторинг подсистемы, которая по показаниям последних отчетов сервера CI имеет слишком много сдублированного кода. Перед тем как передать внесенные изменения в хранилище Subversion, он выполняет *закрытое построение* (private build), в ходе которого компилируется новый исходный код и выполняется

проверка его модулей. После этого он передает изменения со своей машины в хранилище Subversion. Все это время сервер CI CruiseControl опрашивает хранилище Subversion. Через несколько минут сервер CI обнаруживает переданные Тимом изменения и выполняет интеграционное построение, которое, в свою очередь, запускает инструмент автоматизированной инспекции, проверяющий весь код на соответствие стандартам программирования. Тим получает по электронной почте сообщение о несоответствии кода стандарту, быстро вносит изменения и снова передает исходный код в хранилище Subversion. Сервер CI запускает следующее построение, на этот раз проходящее успешно. Просматривая Web-отчеты, созданные сервером CI, Тим обнаруживает, что недавний рефакторинг кода успешно уменьшил объем сдублированного кода в его подсистеме.

Чуть позже в кабинет Тима заходит другой разработчик проекта, Лайза.

Лайза. Послушайте, те изменения, которые вы внесли сегодня утром, нарушили последнее построение!

Тим. Хмм ... но, я запускал проверку.

Лайза. О, а я не успела ее написать.

Тим. А следуете ли вы правилам разработки кода, которые мы установили для проекта?

Исходя из приведенного выше диалога, можно прийти к выводу, что интеграционное построение потерпит неудачу, если код покрыт проверками менее чем на 85 %. Впоследствии Лайза написала проверку для проблемы, которую она упомянула в диалоге с Тимом, нашла дефект и устранила его. Далее интеграционное построение проходит без ошибок.

Принятые термины

Автоматизированный (automated) — процесс без "ручного вмешательства". Как только *полностью автоматизированный* (fully automated) процесс запускается, никакого вмешательства пользователя уже не нужно. Системные администраторы называют его "безголовым" (headless) процессом.

Построение (build) — набор действий, выполняемых при компиляции, проверке, инспекции и развертывании программного обеспечения.

Непрерывный (continuous). Технически термин *непрерывный* означает нечто запущенное один раз и никогда не останавливающееся. Это означало бы, что построение выполняется все время; но это не так. Термин *непрерывный* в контексте CI и в случае серверов CI — это скорее циклически повторяемый процесс опроса хранилища с контролем версий на предмет изменений. Если сервер CI обнаруживает изменения, он запускает сценарий построения.

Непрерывная интеграция (Continuous Integration — CI) — "это способ разработки программного обеспечения, при котором все участники группы осуществляют частую интеграцию результатов своей работы. Обычно каждый человек выполняет интеграцию ежедневно, что приводит к нескольким интеграциям в день. Результат каждой интеграции автоматически проверяется на предмет ошибок по возможности быстрее. Большинство групп находит, что данный подход ведет к значительному уменьшению количества проблем интеграции и позволяет ускорить разработку связного программного обеспечения."[1]

Среда разработки (development environment) — система, в которой разрабатывается программное обеспечение. Сюда может относиться IDE, сценарии построения, инструменты, библиотеки сторонних производителей, серверы и файлы конфигурации.

Инспекция (inspection) — анализ исходного и бинарного кодов по внутренним критериям качества. В контексте этой книги под *инспекцией программного обеспечения* (software inspection) подразумеваются автоматизированные аспекты (статический анализ и анализ времени выполнения).

[1] Источник: `www.martinfowler.com/articles/continuousIntegration.html`.

Интеграция (integration) — действие по объединению отдельных артефактов исходного кода вместе, позволяющее проверить их совместную работу.

Интеграционное построение (integration build) — действие по объединению компонентов (программы и файлов) в систему программного обеспечения. В больших проектах — это объединение нескольких составляющих или только низкоуровневых откомпилированных файлов исходного кода, как в меньших проектах. В повседневной жизни термины *построение* (build) и *интеграционное построение* (integration build) используются как синонимы, но в этой книге мы их различаем, подразумевая, что интеграционное построение осуществляется на отдельной машине интеграционного построения.

Закрытое (системное) построение (private (system) build). Построение, выполняемое локально на рабочей станции разработчика перед передачей изменений в хранилище с контролем версий для уменьшения вероятности того, что последние изменения нарушат интеграционное построение.[2]

Качество (quality). Бесплатный сетевой словарь компьютерной лексики[3] определяет качество как "существенный и специфический атрибут чего-либо ..." и "высшая степень". Термином *качество* зачастую злоупотребляют, и некоторым уже кажется, что это нечто субъективное. Но в данной книге мы придерживаемся мнения, что качество — измеряемая спецификация, точно так же как и любая другая. Это значит, что вы можете указать определенные показатели качества, такие как ремонтопригодность, расширяемость, защищенность, производительность и удобочитаемость.

Финальное построение (release build) — подготовка программного обеспечения для передачи пользователям. Может осуществляться в конце итерации или некоторого другого промежуточного этапа и должно включать все приемочные испытания, а возможно, и другие проверки, например производительности и загруженности.

Риск (risk) — потенциальная возможность возникновения проблемы. Риск, который оправдался, называется *проблемой* (problem). В этой книге основное внимание уделено высокоприоритетным рискам (наносящим наибольший ущерб или имеющим самую высокую вероятность проявления).

Проверка (testing) — общий процесс проверки работоспособности разработанного программного обеспечения. Кроме того, мы разделяем проверки на несколько категорий, включая проверку модуля, проверку компонента и проверку системы. Все, что проверяется (объекты, пакеты, модули и система), и составляет программное обеспечение. Существует множество других типов проверок, например проверка функций и нагрузки, но с точки зрения CI все проверки модуля, написанные разработчиками, выполняются как минимум при построении (хотя процесс построения может быть организован так, что сначала осуществляется быстрая предварительная проверка, сопровождаемая более продолжительными проверками).

В чем значение CI?

На высоком уровне CI имеет следующие преимущества:

- снижение риска;
- уменьшение количества повторяемых процессов, выполняемых вручную;
- построение развертываемого программного обеспечения в любой момент, в любом месте;

[2] На основании книги Стивена Беркзука (Stephen Berczuk) и Бреда Апплетона (Brad Appleton) *Software Configuration Management Patterns.*

[3] См. `www.thefreedictionary.com`.

- обеспечение лучшего контроля проекта;
- повышение доверия к программному продукту со стороны группы разработки.

Давайте рассмотрим, что означают эти принципы и какое значение они имеют.

Снижение риска

Осуществляя интеграцию по нескольку раз в день, вы можете снизить риск в вашем проекте. Это облегчает обнаружение дефектов и контроль состояния программного обеспечения, а также уменьшает количество предположений.

- *Дефекты обнаруживаются и устраняются быстрее.* Поскольку CI интегрирует, а также выполняет проверки и инспекции по нескольку раз в день, существует большая вероятность того, что дефекты обнаруживаются сразу (т.е. когда код проверяется в хранилище с контролем версий), а не при проверке на последнем этапе.
- *Контроль состояния проекта программного обеспечения.* Применение непрерывной проверки и инспекции в автоматизированном процессе интеграции позволяет отслеживать атрибуты состояния программного продукта (например, сложность) через определенное время.
- *Снижение количества предположений.* Перестраивая и проверяя программное обеспечение в соответствующей системе, а также используя тот же самый процесс и сценарии на постоянной основе, вы можете уменьшить количество предположений (например, стоит ли полагаться на библиотеки стороннего производителя или переменные окружения).

Непрерывная интеграция предоставляет страховочную сеть, уменьшающую риск проявления дефектов в базовом коде. Ниже перечислены некоторые из рисков, снизить которые позволяет CI. Эти и другие риски мы обсудим в следующей главе.

- Недостаток связности развертываемого программного обеспечения.
- Позднее обнаружение дефектов.
- Низкое качество программного обеспечения.
- Недостаток контролируемости проекта.

Уменьшение количества повторяемых процессов

Уменьшение количества повторяемых процессов экономит время, деньги и силы. Звучит понятно, не так ли? Повторяемые процессы могут встречаться во всех действиях проекта, включая компиляцию кода, интеграцию базы данных, проверку, инспекцию, развертывание и обратную связь. Автоматизируя CI, вы получаете великолепную возможность обеспечить следующее.

- Процесс *каждый раз* выполняется одинаково.
- Поддерживается упорядоченность процесса. Например, в своих сценариях построения вы можете проводить инспекции (статический анализ) перед проверками.
- Процессы осуществляются каждый раз, когда в хранилище с контролем версий происходит изменение.

Это облегчает следующее:

- снижение трудозатрат на повторяемые процессы высвобождает людей для более интеллектуальных и важных работ;
- возможность преодолевать сопротивление (включая других участников группы), чтобы реализовать преимущества, используя автоматизированные механизмы для важных процессов, таких как проверка и интеграция базы данных.

Построение развертываемого программного обеспечения

Непрерывная интеграция может обеспечить вам выпуск развертываемого программного обеспечения в *любой момент*. С другой стороны, это наиболее очевидное преимущество CI. Мы можем бесконечно говорить об улучшении качества программного обеспечения и снижении риска, но для "посторонних" (клиентов и пользователей) реально развертываемое программное обеспечение значительно важней. Трудно переоценить *важность* данного аргумента. Используя CI, вы вносите незначительные изменения в исходный код и интегрируете их с остальной частью кода на регулярном основании. Если возникнут какие-либо проблемы, участники проекта будут проинформированы об этом и *немедленно* их устранят. В проектах, где подобная практика не используется, проверка программного обеспечения происходит в самом конце и проблемы могут обнаружиться непосредственно перед передачей заказчику. Это может задержать выпуск на время устранения обнаруженных дефектов, затем на время проверки и устранения новых дефектов и т.д., поэтому, поторопившись, можно существенно затянуть конец проекта.

Обеспечение лучшего контроля проекта

CI позволяет лучше замечать тенденции и принимать эффективные решения, а это помогает набраться мужества для введения новых усовершенствований. Когда не получаешь реальных и своевременных данных, необходимых для принятия решений, каждый делает более или менее точные предположения, а проект страдает. Как правило, участники проекта собирают такую информацию вручную, прикладывая обременительные и запоздалые усилия. В результате необходимая информация зачастую не бывает собрана. Применение CI предоставляет следующие преимущества.

- *Эффективные решения*. Система CI может предоставить своевременную информацию о текущем состоянии и качественных показателях построения. Некоторые системы CI могут также отображать частоту дефектов и демонстрировать ход их устранения.
- *Отслеживание тенденций*. Поскольку в системе CI интеграция происходит часто, появляется возможность отслеживать тенденции успеха и отказа построения, общего качества и другой информации о проекте.

Повышение доверия к программному продукту

В целом эффективное применение практик CI может обеспечить больше доверия к созданному программному продукту. При каждом построении ваша группа узнает, как прошли проверки программного обеспечения, каково его поведение, соблюдаются ли стандарты программирования и проектирования, а также какие функции получены в результате.

Без частых интеграций некоторые участники группы могут чувствовать себя неуверенно, поскольку они не знают, повлияли ли их изменения на общий код. Так как система CI может оповестить вас, если что пойдет не так, разработчики и другие участники группы

имеют больше доверия к вносимым изменениям. Кроме того, поскольку система CI предоставляет единое хранилище для всех составляющих программного обеспечения, доверия к его точности также будет больше.

Что может помешать группе использовать CI?

Если CI имеет так много преимуществ, то что может помешать группе разработки использовать ее в своих проектах? Зачастую это комбинация обстоятельств.

- *Увеличение дополнительных затрат на поддержку системы CI.* Как правило, это вовсе не так, поскольку необходимость интегрировать, проверять, инспектировать и развертывать существует независимо от того, используете вы CI или нет. *Поддерживать надежную систему CI значительно проще, чем контролировать процессы, выполняемые вручную.* Как ни странно, но участники сложных многоплатформенных проектов, где потребность в CI наибольшая, как правило, чаще всего сопротивляются ее внедрению, мол, "слишком много дополнительной работы".
- *Слишком много изменений.* Некоторые полагают, что внедрение CI в текущие проекты повлечет за собой изменения слишком многих процессов. Наиболее эффективен инкрементный способ перехода на CI: сначала построение и проверки проводятся реже (например, ежедневно), пока не привыкнут все, а затем их частота увеличивается.
- *Слишком много неудачных построений.* Как правило, это происходит тогда, когда разработчики не выполняют закрытого построения до передачи кода в хранилище с контролем версий. Разработчик может также забыть проверить файл или передать файл, проверка которого не прошла успешно. При использовании CI, из-за частых изменений, быстрый ответ обязателен.
- *Дополнительные издержки на аппаратные средства и программное обеспечение.* На самом деле при внедрении CI придется приобрести отдельную машину для интеграции, но это вполне оправданный расход по сравнению с последующими, более дорогими издержками на поиск и устранение проблем.
- *Эти действия должны будут выполнять разработчики.* Иногда руководство считает, что CI только дублирует действия, которые разработчики должны все равно выполнять. Да, разработчики должны выполнить некоторые из этих действий, но они должны это сделать *более эффективно и надежно в отдельной среде.* Применение автоматизированных инструментальных средств позволяет повысить эффективность и частоту таких действий. Кроме того, это гарантирует, что подобные действия осуществляются в чистой среде, что уменьшает количество предположений и способствует принятию лучших решений.

Как добиться "непрерывной" интеграции?

Иногда просто удивляешься, изучая уровень автоматизации большинства организаций по разработке программного обеспечения. Разработчики тратят колоссальное количество времени на автоматизацию процессов для своих пользователей, но не всегда находят способы автоматизировать свои собственные процессы разработки. Иногда группы полагают, что их работа автоматизирована достаточно, поскольку они написали несколько сценариев, упрощающих ряд этапов процесса разработки. Вот типичный случай.

Джоан (разработчик). ... Я уже автоматизировала это. Я написала несколько пакетных сценариев, которые удаляют и вновь создают таблицы базы данных.

Сью (технический руководитель). Прекрасно. Вы применили это к хранилищу CVS?

Джоан. Нет.

Сью. Вы сделали это частью сценария построения?

Джоан. Нет.

Сью. Ну, если это не часть системы CI, то и автоматизировано не на самом деле... не так ли?

Непрерывная интеграция — это не только процесс объединения нескольких сценариев и их циклического запуска. В приведенном выше примере хорошо то, что Джоан написала сценарии автоматизации, но чтобы они приобрели реальное значение для конечного продукта, их следует применить к хранилищу с контролем версий и сделать частью рабочего процесса построения. Рис. 2.2 иллюстрирует этапы, позволяющие сделать процесс непрерывным.

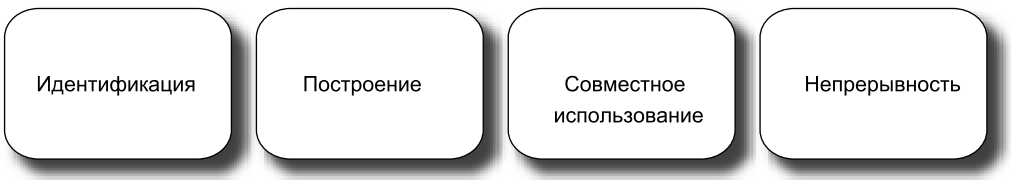


*Рис. 2.2. Переход к CI — "**I B**uild **S**o **C**onsistently" (я строю значит непрерывно)*

Эти этапы могут быть последовательно применены практически к каждому действию, осуществляемому в проекте.

- *Идентификация* (identify). Идентификация — это процесс, который требует автоматизации. Он может осуществляться при компиляции, проверке, инспекции, развертывании, интеграции базы данных и т.д.
- *Построение* (build). Применение сценария построения делает автоматизацию воспроизводимой и однозначной. Сценарии построения можно написать при помощи NAnt для платформ .NET, Ant для платформы Java и Rake для Ruby (это лишь несколько из примеров).
- *Совместное использование* (share). Используя такую систему контроля версий, как Subversion, вы позволяете применять эти сценарии и программы другим. Теперь значение сценария распространяется на весь проект.
- *Непрерывность* (continuous). Следует гарантировать, что автоматизированный процесс будет запущен при внесении любого изменения. Для этого используется сервер CI. Если в вашей группе соблюдается жесткая дисциплина, вы конечно можете решить выполнять построение вручную при каждом изменении, передаваемом системе контроля версий.

Есть сокращение, помогающее запомнить состав и последовательность этих этапов: "**I B**uild **S**o **C**onsistently" (я строю значит непрерывно) — **I**dentify (Идентификация), **B**uild (Построение), **S**hare (Совместное использование) и **C**ontinuous (Непрерывность).

Стремитесь к постепенному росту системы CI. Так ее проще реализовать: по мере добавления каждого нового элемента активность группы растет и на основании этого вы можете лучше спланировать, что необходимо делать дальше. Зачастую попытка внедрить всю систему CI сразу может оказаться плохим решением, точно так же как и рефакторинг

большого количества кода сразу. Сначала добейтесь работоспособности системы, подождите, пока разработчики привыкнут к ней, а затем добавляйте другие автоматизированные процессы по мере необходимости и на основании рисков для проекта.

Это непрерывная компиляция или непрерывная интеграция?

При реализации CI я работал со многими организациями и неоднократно слышал ответ: "Да, мы применяем CI". "Прекрасно!" — думал я, а затем задавал несколько вопросов. Сколько кода покрывают ваши проверки? Как часто и насколько долго выполняется построение? Какова средняя сложность вашего кода? Сколько у вас сдублированного кода? Помечаете ли вы версии построения в хранилище с контролем версий? Где вы храните ваше развертываемое программное обеспечение?

В результате, как правило, оказывается, что все они практикуют скорее "непрерывную компиляцию" с использованием такого инструмента, как CruiseControl, опрашивающего хранилище с контролем версий (например, CVS) на предмет изменений. Обнаружив изменения, он извлекает исходный код из CVS, компилирует его и посылает по электронной почте сообщение, если что-то пошло не так. Система автоматической компиляции программного обеспечения на отдельной машине — это лучше, чем вообще ничего, однако всех преимуществ полноценной системы CI она не обеспечит.

Когда и как следует реализовывать CI?

Лучшее реализовать CI в проекте пораньше. Поскольку позднее сотрудники, скорее всего, окажутся очень заняты и будут сопротивляться нововведениям. Если вы реализуете CI в уже разрабатываемом проекте, то следует начинать с небольших изменений и постепенно наращивать их.

Существуют различные способы установки системы CI. Хотя в конечном счете вы хотите, чтобы построение осуществлялось при каждом изменении, начинать можно с *ежедневного* построения, чтобы ваша организация привыкла к такой практике. Помните: *CI — это не только техническая реализация, но также организационная и культурная*. Люди часто противятся изменениям, и наилучший подход для организации может заключаться в поэтапном введении автоматизированных механизмов.

Сначала построение может подразумевать только компиляцию исходного кода и пакетирование бинарных файлов без автоматизированных проверок. Это может оказаться эффективным для начала, если разработчики не знакомы с автоматизированными инструментами проверки. Если все в порядке и разработчики изучили инструмент проверки, можно двигаться к следующему преимуществу CI: выполнению всех проверок (и инспекций) при каждом изменении.

Развитие интеграции

Действительно ли непрерывная интеграция является новейшим подходом в разработке программного обеспечения? Едва ли. CI — это просто очередное усовершенствование в процессе развития интеграции ПО. Пока программы состояли из нескольких маленьких файлов, их интеграция в систему не имела особых проблем. На протяжении последних лет наилучшей практикой считалось проведение построений ночью. Подобные практики не раз обсуждались в других книгах и статьях. В книге *Microsoft Secrets* Майкла А. Кусумано (Michael A. Cusumano) и Ричарда У. Селби (Richard W. Selby) рассматривается практика ежедневных построений в корпорации Microsoft. Стив Мак-Коннелл (Steve McConnell)

в книге *Software Project Survival Guide* анализирует практику "ежедневного построения и без дымового тестирования (smoke test)" как часть рабочего проекта программного обеспечения.

В книге *Object Solutions: Managing the Object-Oriented Project* Гради Буч (Grady Booch) пишет: "На макроуровне процесс объектно-ориентированной разработки — это один из элементов «непрерывной интеграции»? Регулярный запуск процесса «непрерывной интеграции» позволяет получить работоспособные выпуски, функциональные возможности которых наращиваются при каждом выпуске... Получаемые при этом промежуточные отчеты позволяют руководству отслеживать прогресс и качество, а следовательно, упреждать, выявлять и своевременно устранять риски на постоянной основе." С появлением XP и других гибких (Agile) методологий, а также практик, рекомендуемых CI, люди начали воспринимать концепции не только ежедневного, но и "непрерывного" построения.

Практика CI продолжает развиваться. Вы найдете ее элементы практически в каждой книге по XP. Зачастую при обсуждении практики CI ссылаются на оригинальную статью Мартина Фаулера "Continuous:Integration".[4]

Поскольку аппаратные средства и ресурсы программного обеспечения продолжают совершенствоваться, все больше процессов становится частью того, что составляет непрерывную интеграцию.

Как CI дополняет другие практики разработки?

Практика CI дополняет другие способы разработки программного обеспечения, такие как проверки разработчика, соблюдение стандартов программирования, рефакторинг и малые выпуски. Это не имеет значения, если вы пользуетесь RUP, XP, RUP с XP, SCRUM, Crystal или любой другой методологией. Ниже приведен список, демонстрирующий, как CI дополняет и улучшает эти практики.

- *Проверка разработчика* (developer testing). Разработчики, пишущие проверки, как правило, используют некую среду на базе xUnit, например JUnit или NUnit. Эти проверки могут быть автоматически выполнены из сценариев построения. Поскольку практика CI подразумевает, что построение может быть запущено в любой момент, когда в программное обеспечение внесено изменение, а также то, что автоматизированные проверки являются частью процесса построения, она гарантирует, что автоматизированные *регрессионные проверки* (regression test) будут выполнены для всего базового кода при каждом изменении.
- *Соблюдение стандартов программирования*. Стандарт программирования — это набор правил, которых разработчики должны жестко придерживаться в проекте. Во многих проектах обеспечение соблюдения стандартов — это в значительной степени ручной процесс, подразумевающий просмотр кода. Для оповещения о соблюдении стандартов программирования CI в ходе выполнения сценария построения запускает комплект автоматизированных инструментов статического анализа, которые проверяют исходный код на соответствие установленным стандартам при внесении любых изменений.
- *Рефакторинг* (refactoring)[5]. По утверждению Фаулера, рефакторинг — это "процесс изменения системы программного обеспечения таким способом, который не изме-

[4] См. `www.martinfowler.com/articles/continuousIntegration.html`.

[5] Или реорганизация. — *Примеч. ред.*

няет внешнее поведение кода, однако улучшает его внутреннюю структуру"[6]. Кроме всего прочего, это упрощает код и облегчает его поддержку. Непрерывная интеграция может помочь в рефакторинге, запуская инструменты инспекции, которые выявляют потенциальные проблемные области при каждом построении.

- *Малые выпуски*. Данная практика позволяет испытателям и пользователям получать работоспособное программное обеспечение для опробования и применения так часто, как нужно. CI очень хорошо работает с этой практикой, поскольку интеграция программного обеспечения осуществляется по много раз в день и выпуск доступен практически *в любое время*. Как только система CI установлена, выпуск может быть получен с минимальными усилиями.
- *Коллективная собственность*. Каждый разработчик может работать над любой частью системы программного обеспечения. Это предотвращает "узурпацию знаний", когда только одному человеку известно, что именно содержит и как работает определенная область системы. Практика CI может помочь с коллективной собственностью, гарантируя соответствие стандартам программирования и выполнение регрессионных проверок на непрерывном основании.

Как долго устанавливать систему CI?

Реализация *базовой* системы CI наряду с простыми сценариями построения для нового проекта может занять у вас несколько часов, включая установку и настройку (или немного больше при отсутствии готовых сценариев построения). По мере роста ваших знаний о системе CI будет расти и количество дополнительных инструментов инспекции, развертывать которые сложней, но они обеспечивают более полную проверку, и много других процессов. Эти дополнительные средства, как правило, устанавливают не сразу.

В уже выполняемом проекте установка системы CI может занять дни, недели или даже месяцы. Это зависит также от людей, работающих над проектом. Как правило, при переходе на непрерывную и автоматизированную систему, например с использованием сервера CI, приходится решать множество задач. Иногда происходит переход с пакетных файлов (bat) или сценариев оболочки на инструменты сценариев построения типа Ant, управляющие бинарными зависимостями всего проекта. В других случаях вы ранее, возможно, использовали для интеграции и развертывания ваш IDE. Так или иначе, но путь к полному восприятию CI может оказаться немного дольше.

CI и вы

Для эффективной работы CI в проекте разработчики должны изменить свои повседневные привычки создания программного обеспечения. Они должны передавать код чаще, сделать своей приоритетной задачей устранение ошибок построения, осуществлять автоматизированное построение с проверками, которые всегда должны быть успешными на 100 %, не получать и не передавать сбойный код в хранилище с контролем версий.

Практики, которые мы рекомендуем, требуют некоторой дисциплины, однако обеспечивают преимущества, описанные в этой главе. Наилучшая ситуация, когда большинство участников проекта убеждены, что практики CI окупятся многократно.

---

[6] Фаулер и др. *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).

Вот семь практик, которые мы считаем подходящими для лиц и групп, реализующих в проекте CI.

- Передавайте код часто.
- Не передавайте сбойный код.
- Ликвидируйте проблемы построения немедленно.
- Пишите автоматизированные проверки разработки.
- Все проверки и инспекции должны быть пройдены.
- Выполняйте закрытое построение.
- Избегайте получения сбойного кода.

Следующие разделы рассматривают каждую практику более подробно.

Передавайте код часто

Одним из основных принципов CI является *ранняя и частая* интеграция. Чтобы ощутить преимущества CI, разработчики должны передавать код часто.

Ожидание передачи кода в хранилище с контролем версий больше одного дня делает интеграцию слишком долгой и может помешать разработчикам использовать последние изменения. Чтобы чаще передавать код, попробуйте применить один или оба следующих метода.

- *Делайте небольшие изменения.* Старайтесь не изменять слишком много компонентов сразу. Выберите небольшую задачу, напишите исходный код и проверки для него, выполните проверки, а затем передавайте код в хранилище с контролем версий.
- *Передавайте после решения каждой задачи.* С учетом разделения задач (работ) на фрагменты, которые могут быть реализованы за несколько часов, некоторые компании требуют, чтобы разработчики передавали свой код по мере выполнения каждой задачи.

Не нужно добиваться того, чтобы все передавали код обязательно каждый день. Вы быстро обнаружите, что самые серьезные ошибки построения чаще всего возникают из-за конфликтов между изменениями. Люди склонны ошибаться именно в конце дня, когда, устав, торопятся домой. Чем дольше вы ждете интеграции, тем трудней она оказывается.

Я просто не могу передавать

Мой друг выполняет проект с участием 25 разработчиков и хотел бы применить практики CI, однако не может добиться от коллег, чтобы те передавали код часто. Я выяснил, что основная тому причина заключается в “национальных” особенностях проекта. Иногда разработчики не хотят передавать свой код, пока он не станет “совершенным”. Обычно это происходит потому, что их изменения влияют на слишком многие компоненты. Однако единственный эффективный способ реализации CI подразумевает частую передачу кода в хранилище с контролем версий, а это означает, что все разработчики должны поддерживать данную практику разработки, охватывая небольшие фрагменты кода и разделяя свои задачи на меньшие элементы.

Не передавайте сбойный код

Опаснейшим предположением для проекта является то, что каждый его участник знает о запрете передавать неработоспособный код в хранилище с контролем версий. Для решительного снижения этого риска необходим хорошо продуманный сценарий построения, который компилирует и проверяет код воспроизводимым способом. Введите в обычай группы разработки непременное выполнение закрытого построения (которое очень похоже на процесс интеграционного построения) перед передачей кода в хранилище с контролем версий. Более подробная информация и рекомендации по этой теме приведены далее в разделе “Выполняйте закрытое построение”.

Ликвидируйте проблемы построения немедленно

Сбойное построение (broken build) — это построение, потерпевшее неудачу. Причиной может быть ошибка компиляции, неуспех проверки или инспекции, проблема с базой данных или неудача при развертывании. В системе CI данные проблемы следует устранять немедленно; к счастью, в инкрементной среде каждая ошибка обнаруживается быстро, а следовательно, ее пропуск менее вероятен. В некоторых проектах предусмотрен штраф за сбой при построении, например удержание некоторых денег или размещение изображения разработчика, по вине которого произошел сбой, на большом мониторе компании (шутка, конечно, на самом деле так никто не делает). Культура проектирования подразумевает, что устранение ошибок построения — это главный приоритет. Таким образом, не только некоторые участники группы, а и все участники проекта должны иметь возможность вернуться к прежней версии своей работы.

Пишите автоматизированные проверки разработки

Построение должно быть полностью автоматизировано. Для запуска проверок в системе CI их также следует автоматизировать. Написание проверок в среде xUnit, такой как NUnit или JUnit, обеспечит возможность их выполнения в автоматическом режиме. Более подробная информация о написании автоматизированных проверок приведена в главе 6, “Непрерывная проверка”.

Все проверки и инспекции должны быть пройдены

В среде CI для построения должно быть успешно осуществлено 100 % автоматизированных проверок (это технический критерий, не ожидайте, что все сотрудники и все их работы будут совершенны). Последние столь же важны, как и компиляция. Просто каждый добивается того, чтобы его код не только компилировался, но и безошибочно работал; а тот код, в котором при проверке были выявлены ошибки, не будет работать правильно. Принятие кода, который не прошел проверку, может привести к снижению качества программного обеспечения.

Недобросовестный разработчик может просто закомментировать проверку, которую не проходит его код. Конечно, это вредит общей цели. *Инструменты покрытия* (coverage tool) позволяют выявить исходный код, не имеющий соответствующей проверки. Вы можете запускать инструмент покрытия кода как часть интеграционного построения.

То же относится и к выполнению автоматизированных инспекций программного обеспечения. Используйте набор общих правил программирования и выработайте стандарты, которым должен удовлетворять весь код. Впоследствии могут быть добавлены расширенные инспекции, которые не приводят к неудаче построения, а позволяют выявить области кода, требующие более внимательного исследования.

Выполняйте закрытое построение

Для предотвращения сбоев при построении разработчики должны имитировать интеграционное построение на своей локальной рабочей станции после завершения проверок модуля. Такая операция позволит интегрировать новое работоспособное программное обеспечение с работоспособным программным обеспечением ото всех других разработчиков[7], полученным из хранилища, причем *с учетом* последних изменений, внесенных в него. Таким образом снижается вероятность того, что переданный разработчиком код потерпит неудачу на сервере интеграционного построения.

Удерживайте построение в состоянии “зеленый”

На мой взгляд, существуют две меры эффективности использования CI: количество передач и состояние построения. Каждый разработчик должен осуществлять передачу в хранилище по крайней мере каждый день, и количество передач обычно демонстрирует размер изменений (частые передачи, как правило, свидетельствуют о небольших изменениях, и это хорошо). Чаще всего состоянием построения будет “зеленый” (успешно); возьмите это за правило для своей группы. Но все мы иногда получаем состояние построения “красный”; в этом случае очень важно как можно скорее вернуть его в состояние “зеленый”. Никогда не позволяйте группе привыкать к состоянию “красный”, пока она ожидает завершения той или иной задачи проекта. Длительное пребывание в состоянии “красный” вредит другим критериям проекта и ликвидирует многие преимущества применения CI.

Избегайте получения сбойного кода

При сбое построения не извлекайте последний код из хранилища с контролем версий. В противном случае для продолжения работы (компиляции и проверки своего кода) вам придется потратить много времени на поиск чужой ошибки, приведшей к неудаче построения. Безусловно, устранение ошибки — задача всей группы, но ответственные за нее разработчики *уже*, скорее всего, заняты ее устранением и по завершении передадут свой код в хранилище с контролем версий. Однако разработчик может и не прочитать сообщение электронной почты о сбойном построении. Вот когда пассивный механизм обратной связи типа световой или звуковой сигнализации может быть полезен для оповещения сотрудников. Мы считаем критически важным, чтобы все разработчики знали о состоянии кода

[7] Некоторые инструменты управления конфигурацией, такие как ClearCase, способны автоматически обновлять локальную среду изменениями из хранилища с контролем версий (в ClearCase это называется “dynamic views” (динамические представления)).

в хранилище с контролем версий. Более подробная информация о механизмах непрерывной обратной связи приведена в главе 9, “Непрерывная обратная связь”. Альтернативным, но не рекомендуемым подходом избежания отладки является использование системы контроля версий для отката любых изменений, внесенных с момента последней передачи.

Резюме

Теперь у вас достаточно аргументов в пользу CI для разговоров о ней с другими. В этой главе рассматривались некоторые из основ CI, способы обеспечения непрерывности процесса и ряд других тем, подробно обсуждаемых в последующих главах. В табл. 2.1 приведены семь практик, которые имеет смысл использовать при непрерывной интеграции. В следующей главе рассматриваются риски для программного обеспечения, которые способна снижать CI, улучшая тем самым качество.

Таблица 2.1. Практики CI, обсуждавшиеся в этой главе

Практика	Описание
Передавайте код часто	Передавайте код в хранилище с контролем версий по крайней мере один раз в день
Не передавайте сбойный код	Не передавайте код, который не компилируется с другим кодом или не проходит проверок
Ликвидируйте проблемы построения немедленно	Хотя это ответственность всей группы, при неудачном построении исправлять ошибки должен разработчик, последним передавший код
Пишите автоматизированные проверки	Удостоверьтесь, что ваше программное обеспечение использует автоматизированные проверки разработчика. Запускайте их в ходе автоматизированного построения с использованием CI
Все проверки и инспекции должны быть пройдены	Не 90 % и не 95 %, а все проверки должны пройти успешно перед передачей кода в хранилище с контролем версий
Выполняйте закрытое построение	Чтобы предотвратить отказ интеграции, получите изменения от других разработчиков и последние изменения из хранилища, а затем выполните полное интеграционное построение локально (это называется также закрытым построением)
Избегайте получения сбойного кода	Если построение потерпело неудачу, вы только потеряете время, если получите код из хранилища. Подождите следующего изменения или помогите разработчикам устранить причину отказа построения, а затем получите последнюю версию кода

Вопросы

Реализация CI — это гораздо больше, чем установка и настройка некоторых инструментальных средств. Сколько из указанных ниже операций вы постоянно выполняете в вашем проекте? Сколько из других практик CI способно улучшить ваши возможности разработки?

- Передает ли каждый участник вашей группы код по крайней мере один раз в день? Используете ли вы методы, поощряющие частую передачу кода?
- Какой процент ежедневных интеграционных построений оказывается успешным (т.е. проходит все проверки самое последнее построение)?

- Каждый ли в вашей группе выполняет закрытое построение перед передачей кода в хранилище, чтобы уменьшить вероятность ошибки интеграции?
- Написан ли сценарий вашего построения так, чтобы он считал неудачей сбой при любой из проверок или инспекций?
- Действительно ли устранение сбоя интеграционного построения является приоритетом ваших проектов?
- Избегаете ли вы получения последнего кода из системы контроля версий при сбое построения?
- Как часто вы добавляете автоматизированные процессы в построение и систему CI, делаете ли это непрерывно или периодически?

Глава 3

Снижение риска с использованием CI

Качество — это когда все сделано правильно, даже если никто не проверяет.

Генри Форд

Проект никогда не проходит без проблем. Занимаясь CI на практике, вы неизбежно выясняете, что все этапы процесса разработки рано или поздно повторяются. CI помогает выявить и снизить возможные риски, упрощая оценку и оповещение о состоянии проекта на основании конкретных доказательств. Сколько программного обеспечения мы реализовали? Ответ: посмотрите последнее построение. Каково покрытия кода проверками? Ответ: посмотрите последнее построение. Кто отличился в последнем коде? Ответ: посмотрите последнее построение.

В этой главе рассматриваются риски, которые может смягчать CI, такие, например, как позднее обнаружение дефектов, недостаточный контроль над проектом, низкое качество программного обеспечения и неспособность создать развертываемое программное обеспечение.

Все проекты начинаются с благих намерений, но завершаются благополучно далеко не все. Погубившие их проблемы — *результат потери контроля над рисками*. Как уже упоминалось ранее, мы нередко слышим от участников группы разработки: “На мой взгляд, проверки и обзоры кода (объединенные или нет) — плохие практики”. А когда сроки сдачи проекта начинают поджимать, группа обычно отказывается от этих практик в первую очередь. Данная глава посвящена рискам для программного обеспечения, которые может снизить использование различных аспектов CI. Применяя CI, вы можете построить “качественно защищенную сеть” и создать программное обеспечение быстрее. Когда после каждого изменения вы нажимаете “кнопку <Integrate>”, вы закладываете фундамент снижения рисков заранее, как показано на рис. 3.1.

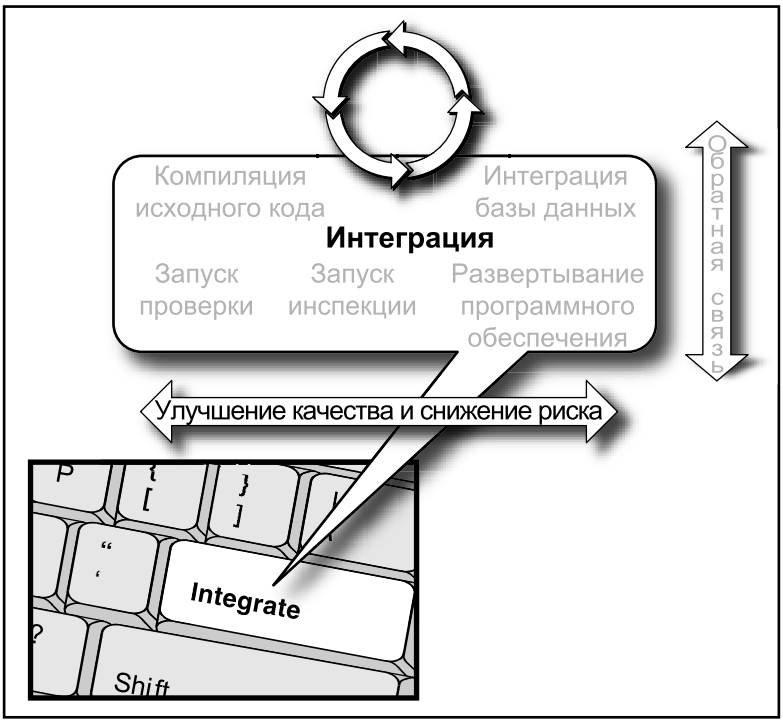


Рис. 3.1. CI может улучшить качество программного обеспечения и снизить риск

Если вам удастся снизить некоторые риски программного обеспечения, вы сможете улучшать его качество. Для описания рисков в этой главе используется следующий шаблон:

- введение и описание *программного риска* (software risk);
- *сценарий* (scenario), основанный на нашем опыте;
- *решение* (solution) по снижению риска с использованием аспекта CI.

Любой проект подвергается множеству рисков, с которыми приходится справляться. Мы сосредоточимся на ключевых рисках, которые вы можете снизить при помощи CI. Безусловно, *непосредственно* CI не решит таких проблем, как выявление бизнес-требований клиента, осознание его производственных нужд, финансирование или управление ресурсами, но использование CI поможет быстрее обнаружить ошибки в разрабатываемом программном обеспечении.

Строя программное обеспечение после каждого изменения, CI может заставить время работать на вас. CI позволит вам сосредоточиться на наиболее серьезных программных проблемах проекта. Поскольку CI — это составная практика, рассматриваемые в этой главе риски охватывают множество методологий разработки программного обеспечения.

- Отсутствие развертываемого программного обеспечения.
- Позднее выявление дефектов.
- Плохой контроль проекта.
- Низкокачественное программное обеспечение.

Вы можете сказать: "О, я не раз слышал обо всех этих рисках, здесь нет ничего нового для меня". Но *знание* о риске не гарантирует возможности *избежать* его. Существуют более эффективные и производительные способы выявления и предотвращения риска, позволяющие не отвлекаясь на них, сосредоточить основное внимание на других вопросах проекта. Подобно большинству практик, эта сводится к эффективной реализации. В последующих главах, используя модель кнопки <Integrate>, мы продемонстрируем эффективные способы выявления и снижения таких рисков.

Риск: отсутствие развертываемого программного обеспечения

Я участвовал в проекте, где мы строили программное обеспечение на отдельной машине каждый месяц или около того. При окончательном построении, когда подошли сроки сдачи программного обеспечения заказчику, большинство участников группы задерживалось на работе до поздней ночи, пытаясь сотворить чудо. На протяжении этого "ада интеграции" мы находили неработающие интерфейсы, пропущенные файлы конфигурации, множество компонентов, выполняющих похожие функции, а также испытывали огромные трудности при объединении многих изменений, внесенных при последнем построении. Из-за этого мы иногда пропускали критически важные промежуточные отчеты в проекте.

В другом проекте интегрирующее построение программного обеспечения было ручным процессом, инициализируемым из IDE. Как правило, мы еженедельно вручную интегрировали программное обеспечение. Иногда для построения программного обеспечения, не располагающегося в хранилище с контролем версий, использовался анализатор *управления конфигурацией* (Configuration Management — CM). Из-за недостаточной автоматизации выполнение построения требовало значительных трудозатрат. Поскольку мы не осуществляли построение на отдельной машине в чистой системе, мы не имели гарантии, что оно проходило правильно. Тремя полученными результатами были следующие:

- низкое доверие к программному обеспечению, даже если его удавалось *построить*;
- продолжительные интеграционные фазы перед выпуском (внутренним (в рамках группы) и внешним (для клиента)), на протяжении которых ничего другого не делалось;
- неспособность производить и воспроизводить *проверочное построение* (testable build).

Сценарий: "На моей машине это работает"

Существует множество причин, по которым группа разработчиков оказывается неспособна создать развертываемое программное обеспечение. Все — от неудачного прохождения проверок до передачи в хранилище с контролем версий не тех файлов — может способствовать сбою построения. Давайте рассмотрим такой случай.

Джон (технический руководитель). У нас проблема с последним построением на сервере проверки.

Адам (разработчик). Это странно, когда я проводил построение на своей машине, все работало. Давайте посмотрим. Вот, все работает.

Джон. Я понял, в чем дело. Вы не передали ваши новые файлы в хранилище Subversion.

Решение

Трудно переоценить важность ликвидации жесткой связи между IDE и процессом построения. Для интеграции программного обеспечения всегда используйте отдельную машину. Удостоверьтесь, чтобы в хранилище с контролем версий было все необходимое для построения программного обеспечения. И наконец, создайте систему CI. Используйте для автоматизированного построения сервер CI, например CruiseControl, наравне с такими инструментальными средствами, как Ant, NAnt или Rake. Сервер CruiseControl отслеживает изменения в хранилище с контролем версий, а обнаружив их, запускает сценарий создания проекта. Вы можете увеличивать возможности системы CI, включив в построение проверки, инспекции и развертывание программного обеспечения в среде разработки и проверки. Таким образом, вы *всегда* будете иметь работоспособное программное обеспечение.

Сценарий: синхронизация с базой данных

Если вы неспособны быстро пересоздать базу данных в ходе разработки, вам будет трудно вносить изменения. Зачастую это связано с разобщенностью группы базы данных и группы разработчиков, т.е. обе группы заняты исключительно собственными обязанностями и не общаются между собой. Как может быть интегрирован продукт, если не интегрированы группы? В подобном случае администратор базы данных, например, не может передать бо́льшую часть сценариев в хранилище с контролем версий. А это может привести к обострению следующих рисков:

- затруднению внесения изменений или рефакторинга базы данных и исходного кода;
- трудностям при заполнении базы наборами проверочных данных;
- трудностям в поддержке сред разработки и проверки (например, Development (Разработка), Integration (Интеграция) и Test (Проверка)).

Это негативно сказывается на разработке, поскольку группа базы данных не успевает за группой разработки и наоборот. Разработчики программного обеспечения и базы данных вообще могут использовать различные версии базы данных. Участники проекта не могут пользоваться единым первоисточником (хранилищем с контролем версий) для получения последней базы данных. Данную проблему иллюстрирует следующий диалог.

Лорен (разработчик). У меня было много проблем с проверкой базы данных v1.2.1.b1 при построении 1345.

Полин (разработчик базы данных). О нет, с 1345 нужно использовать версию v1.2.1.b2, но я должна сначала внести в нее несколько изменений.

Лорен. А я потратила впустую четыре часа.

Полин. Сначала нужно было посоветоваться со мной.

Решение

Данное решение потребовало бы фундаментального изменения некоторых проектов, поскольку подразумевает, что база данных — это не отдельный объект разработки.

- Поместите все артефакты базы данных в *хранилище с контролем версий*. Это означает необходимость полной перестройки схемы базы данных и самих данных, включая сценарии создания базы данных и манипулирования данными, хранимые процедуры, триггеры и все остальные элементы базы данных.

- Организуйте перестройку базы данных и данных из *сценария построения*, удаляя и вновь создавая базу данных и ее таблицы. Добавьте хранимые процедуры и триггеры, а затем, наконец, проверочные данные.
- *Проверьте (и проинспектируйте) базу данных*. Как правило, для этого используются компоненты проверки. В некоторых случаях вы будете вынуждены написать специальные проверки для базы данных.

Более подробная информация о сценариях и решениях приведена в главе 5, "Непрерывная интеграция баз данных".

Сценарий: ошибочный щелчок

Развертывание программного обеспечения вручную — это потеря времени и сил. В одном из проектов мы делали это, используя утилиту Web-администрирования сервера приложения. Предполагалось, что это будет происходить один раз в день, но поскольку группа обычно сталкивалась с различными другими проблемами, развертывание при интеграционном построении, оставаясь напоследок, становилось узким местом. Этот повторяемый, занудный процесс занимал 10–15 минут каждый день при благоприятном исходе. Проблема заключалась в следующем: мы тратили время на то, что должно было осуществляться автоматически (развертывание на машине проверки). Кроме того, было очень просто нажить проблему, щелкнув не на той кнопке инструмента администрирования.

Вот типичной пример проблемы, полученной в результате ручного способа развертывания.

Рэйчел (разработчик). Передано ли последнее построение на сервер разработки? Где Джон?

Келли (разработчик). Джон пошел обедать. Наверное, он передал обновление на сервер.

Рэйчел. Ладно, я подожду, пока он вернется.

Позже, когда Джон пришел...

Рэйчел. Джон, что там с последним построением? Похоже, JSP не был предварительно скомпилирован, и теперь мы получаем ошибки во время выполнения программы.

Джон (технический руководитель). Ой, извините. Я, должно быть, забыл установить этот флажок, когда вчера развертывал приложения с помощью Web-инструмента.

Решение

Мы автоматизировали процесс развертывания в наших проектах, добавив его в сценарий построения Ant, который использует параметры командной строки сервера приложений. Это ликвидировало узкое место при развертывании программного обеспечения и снизило вероятность ошибок. Теперь мы всегда имели последнюю проверяемую версию программного обеспечения. Мы запускали этот сценарий построения Ant на сервере CI CruiseControl каждый раз при передаче изменений в хранилище с контролем версий. Более подробная информация по данной теме приведена в главе 8, "Непрерывное развертывание".

Риск: позднее выявление дефектов

В некоторых проектах мы выполняли проверку вручную. Мы не знали точно, привели ли последние изменения программного обеспечения к возникновению еще каких-то проблем. Например, запоздалый цикл исправления одного дефекта может лишь проявить

другие дефекты. Мы не имели никакого доверия к сделанным изменениям, поскольку не знали достоверно их результата. Не было никакого способа удостовериться, выполняли ли разработчики проверки программного обеспечения, поскольку все это делалось вручную.

Сценарий: регрессионная проверка

Давайте рассмотрим сценарий регрессионной проверки.

Салли (технический руководитель). Я заметила, что последняя версия, развернутая в проверочной среде, имеет ту же ошибку, которая была и два месяца назад. В чем дело?

Кайл (разработчик). Не знаю. Все свои последние изменения я проверил.

Салли. Вы выполняли все остальные проверки для других частей системы?

Кайл. Нет, у меня не было времени проверять *все* вручную. Это, вероятно, произошло потому, что я не нашел эту ошибку прежде.

Решение

В новых проектах мы начали писать проверки модулей и компонентов на JUnit для бизнес-уровня, уровня данных и общие. Для текущих проектов мы писали проверки модулей и кода, который был изменен с учетом вероятных дефектов. Мы настроили сценарии построения Ant так, чтобы выполнять все проверки модулей и публиковать отчет для каждого, кто участвовал в построении.

Ниже приведена последовательность действий, демонстрирующая, как вы можете использовать систему CI для автоматизации регрессионной проверки в вашем проекте.

1. Напишите проверку для всего исходного кода (для начала хорошо подойдет среда xUnit).
2. Запустите проверки из сценария построения (как правило, Ant или NAnt).
3. Запускайте проверки непрерывно, как часть вашей системы CI, чтобы они были применены при каждом изменении в хранилище с контролем версий (с использованием сервера CI CruiseControl или подобного).

Точно так же просто вы можете автоматизировать регрессионную проверку в своем проекте! Более подробная информация о создании проверок как неотъемлемой части построения на всех уровнях приведена в главе 6, “Непрерывная проверка”.

Сценарий: покрытие проверками

Если вы пишете и выполняете проверки, а затем анализируете результаты, то, вероятно, хотите также знать, какое количество кода *фактически* проверяется. Поскольку до внедрения системы CI большинство проверок модулей в проекте осуществлялось вручную, не было никакого независимого способа удостовериться, выполнялись ли проверки на самом деле. Как менеджеру узнать объем фактических проверок? Рассмотрим следующий диалог.

Эвелин (менеджер). Запускали ли вы проверки модуля прежде, чем передать изменения в хранилище?

Ноа (разработчик). Да.

Эвелин. Прекрасно. Насколько они подходят к другим реализованным вами компонентам?

О чем Эвелин *не спросила*? Давайте попробуем снова.

Эвелин. Вы написали новые проверки или обновили уже существующие для вашего нового кода?

Ноа. Да.

Эвелин. Все ли проверки пройдены успешно?

Ноа. Да.

Эвелин. Как вы определили, что адекватно проверено достаточно кода?

Второй набор вопросов немного лучше, но это все еще излишне качественный анализ того, что может быть описано более конкретно, количественно. Давайте перейдем к решению.

Решение

Если разработчики или группы полагают, что они написали подходящие проверки для своего исходного кода, то вы можете запустить инструмент покрытия и оценить объем исходного кода, который фактически охвачен проверками. Большинство инструментов отобразит процент покрытия по пакетам и классам.

Использование CI может гарантировать, что это покрытие проверками всегда будет актуально. Например, вы можете запускать инструмент покрытия проверками как часть сценария построения вашей системы CI всякий раз при внесении изменений в хранилище с контролем версий. Более подробная информация по этой теме приведена в главе 7, "Непрерывная инспекция".

Риск: плохой контроль проекта

Механизмы связи, поддерживаемой вручную, требуют серьезной координации, чтобы гарантировать своевременную доставку информации проекта нужным людям. Безусловно, можно просто повернуться к разработчику рядом и сообщить ему, что последнее построение уже находится на совместно используемом диске. В маленьком коллективе это срабатывает, но не в большом. Что, если другие разработчики, нуждающиеся в данной информации, находятся на перерыве или недоступны по иной причине? Кто оповестит вас, если сервер отключится? Некоторые полагают, что они могут снизить данный риск, посылая электронную почту вручную. Но это не может гарантировать, что отправленная вовремя и нужным людям информация будет получена ими, поскольку они могут временно не иметь доступа к своей электронной почте или попросту забыть проверить ее.

Сценарий: "Вы получали сообщение?"

Существует много различных сценариев для этого риска, вот один из них.

Эвелин (менеджер). Над чем вы работаете, Ноа?

Ноа (испытатель). Я жду последнего построения, чтобы развернуть его и начать проверять.

Эвелин. Последнее построение было развернуто на сервере проверки два дня назад. Разве вы не слышали?

Ноа. Нет, меня не было в офисе несколько дней.

Решение

Чтобы снизить подобный риск, мы установили и настроили на сервере CI CruiseControl автоматизированный механизм рассылки электронной почты всем заинтересованным лицам в случае сбоя построения. Кроме того, мы добавили уведомления SMS, чтобы сотрудники получали текстовые сообщения на своих мобильных телефонах при отсутствии

доступа к электронной почте. Мы установили также автоматизированные агенты, которые регулярно проверяли доступность серверов. Более подробная информация по этой теме и примеры приведены в главе 9, “Непрерывная обратная связь”.

Сценарий: неспособность представить программное обеспечение

В одном из проектов мы осуществляли расширение возможностей и модернизацию существующего программного обеспечения. Но у нас не было никакого инструмента *обратного проектирования* (reverse-engineering), который мог бы представить нам всю картину: модель классов и отношений. При наличии реальной схемы классов нам было бы куда легче выявить повторяющиеся элементы и недостатки структуры, повысив таким образом эффективность решения.

Мейл (разработчик). Привет. Я новый сотрудник проекта, мне хотелось бы просмотреть его. Могу ли я увидеть какие-нибудь схемы UML или другие схемы?

Али (разработчик). Хмм... Мы здесь не используем UML. Все, что вы можете — это прочитать код. Если вы не умеете этого, то, возможно, не приживетесь здесь.

Мейл. Все в порядке; я просто надеялся сразу увидеть общую картину и выяснить архитектуру, а не постепенно вникать в код. Мне больше нравится наглядность.

Решение

Чтобы уменьшить время между проявлением недостатка проекта и его исправлением, мы начали создавать схемы проекта, используя систему CI. Мы применили автоматизированный инструмент документирования кода Doxygen как составную часть системы CI. Инструмент Doxygen документирует исходный код и создает схемы UML, т.е. модель программного обеспечения. Поскольку инструмент запускался в составе CI, мы всегда имели актуальную документацию, созданную на основе программного обеспечения, последним переданного в хранилище с контролем версий.

Хотя мы документировали все с использованием системы CI, мы решили также создать небольшой документ, на одну–две страницы, чтобы описать архитектуру программного обеспечения, отметив ключевые компоненты и интерфейс для новых разработчиков.

Риск: низкокачественное программное обеспечение

Существуют дефекты и *потенциальные* дефекты. Если ваше программное обеспечение не очень хорошо спроектировано, если в нем не соблюдаются стандарты или оно сложно в поддержке, вы можете иметь потенциальные дефекты. Иногда такой код или проект называют *вонючим* (smells) — “симптом, который иногда бывает ошибочным”.[1] Некоторые полагают, что причиной низкого качества программного обеспечения является исключительно недостаточное финансирование проекта. Последнее существенно, но имеется также и ряд других проблем, обусловливающих это. Чрезмерно сложный код, который не соответствует архитектуре, и дублированный код обычно приводят к дефектам в программном обеспечении. Поиск такого кода и его симптомов прежде, чем он превратится в дефекты, может сэкономить много времени и денег, а также повысить качество программного обеспечения. В этом разделе мы исследуем несколько таких сценариев.

[1] См. `http://en.wikipedia.org/wiki/Code_smell`.

В одном из проектов мы понятия не имели, насколько поддерживаемо наше программное обеспечение, хотя вручную осматривали весь исходный код каждый день. Мы не могли выявить качественные тенденции при разработке. Многие участники проекта похоже "не располагали временем" на повышение внутреннего качества программного обеспечения и не знали, с чего начать. Некоторые проекты имели документированный стандарт программирования, который сотрудники редко соблюдали или читали. Другие не имели никакого стандарта вообще. В некоторых из проектов энтропия была очевидна, поскольку мы боялись, что сделанные изменения нарушат программное обеспечение.

Сценарий: соблюдение стандартов программирования

Вот типичный диалог о соблюдении стандартов программирования.

Брайан (разработчик). На мой взгляд, ваш код малопонятен. Вы читали документ о стандарте программирования на 30 страницах, который нам раздали в прошлом месяце?

Линдсей (разработчик). Я использую тот же стиль, что и в предыдущей работе. Мой код довольно сложен, поэтому вам, вероятно, трудно его понять.

Брайан. Написание кода, с которым другие не могут работать, не делает вас умнее других; это делает вас менее ценным сотрудником. Мне ваш код дольше просматривать и модифицировать. Пожалуйста, прочитайте документ о стандартах программирования, как только сможете. Сначала вы можете переделать в соответствии со стандартами ваш прежний код, а затем вернуться к новому коду.

Решение

Вместо того чтобы писать документ о стандартах на 30 страниц, мы создали хорошо аннотированный пример класса, который содержал все стандарты программирования.[2] Мы применили стандарт программирования, задействовав автоматизированные инструменты инспекции как часть сценариев построения, инициализируемых сервером CruiseControl. При работе над проектами Java мы использовали прежде всего инструменты Checkstyle[3] и PMD[4] для выявления всех строк кода, которые не удовлетворяли установленным стандартам. Мы обеспечили представление этой информации в форме отчетов HTML, которые интегрировали на сервере CI CruiseControl. В последних проектах мы не допустили бы построения, если бы имелось хоть какое-то нарушение стандарта программирования.

Сценарий: соответствие архитектуре

Исходный код, не соответствующий проекту, труднее поддерживать. Случалось ли вам участвовать в проекте, который начинался с очень изящной архитектуры программного обеспечения, а затем, к концу, превращался в "Большой Ком Грязи" ("Big Ball of Mud")[5]?

---

[2] См `www.xp123.com/xplor/xp0002f/codingstd.gif` в книге Уильяма К. Вейка (William C. Wake) *Java Coding Conventions on One Page*.

[3] Checkstyle — это инструмент статического анализа, который оценивает исходный код и сообщает о любых отклонениях от установленного стандарта программирования. Доступен на сайте `http://check-style.sourceforge.net/`.

[4] PMD — измерительный инструмент, который сообщает о любых аномалиях в исходном коде, таких как неиспользуемые переменные и неиспользуемые импортируемые элементы, или о чрезмерно сложном коде. Доступен на сайте `http://pmd.sourceforge.net/`.

[5] "Система... которая не имеет никакой реальной архитектуры." См. *`http://en.wikipedia.org/wiki/Big_ball_of_mud`*.

Возможно, архитектор разработал целую систему, используя инструмент моделирования UML, и сказал нечто вроде “Следуйте этой архитектуре!” Это, возможно, крайний случай, но в жизни всегда есть промежуточные оттенки серого.

Различие между задуманной и фактической архитектурой может стать проблемой. Предположим, например, что архитектура постановляет: “Слой данных никогда не должен обращаться к бизнес-уровню”. Возможно, архитектор использовал инструмент моделирования UML для перепроектирования модели на основании этой архитектуры в исходный код. Но через какое-то время код изменяется, и архитектура перестает соответствовать его первоначальному проекту. Предположим, например, что в проект приходит новый разработчик, который находит на бизнес-уровне некие полезные методы и вызывает их со слоя данных. Но это нарушение архитектуры проекта. Как можно гарантировать, что такое не случится?

Джейн (архитектор). Парни, вы соблюдаете архитектуру? Я обнаружила некоторые проблемы в одном из контроллеров, где вы вызываете компонент непосредственно на слое данных.

Марк и Чарли (разработчики). (Озадаченные возгласы).

Джейн. Я создала все эти схемы UML для того, чтобы каждый из вас следовал установленной архитектуре. Вы не соблюдаете протокол, который был установлен месяц назад.

Чарли. Я смотрел архитектуру в начале проекта, но она изменялась с тех пор несколько раз, поэтому нам трудно поддерживать ее на должном уровне.

Решение

Чтобы оценить соблюдение стандартов архитектуры проекта, примените автоматизированные инструменты инспекции. Например, вы могли бы добавить правило, согласно которому классы контроллера никогда не должны непосредственно обращаться к объектам доступа к данным. Для создания отчетов о соблюдении архитектуры можно использовать такие инструменты анализа зависимостей, как JDepend[6] или NDepend. Их можно запускать при каждом интеграционным построении.

Сценарий: сдвоенный код

Сдвоенный код усложняет поддержку и увеличивает издержки. Риск копирования и вставки кода существует практически в каждом проекте. Фактически у большинства широко известных комплектов разработчика и инструментальных средств дублировано более 25 % кода. В одной компании мы проанализировали все рабочие проекты программного обеспечения и обнаружили, что сдублировано порядка 45 % кода. Поскольку все копии кода должны быть одинаковы, их поддержка может оказаться проблематичной. Предположим, например, что система имеет пять одинаковых копий кода в различных подсистемах. Скажем, существует некий код, который проверяет авторизацию зарегистрировавшегося пользователя. Вместо того чтобы написать единый метод, разработчик решил скопировать и вставить код везде, где нужна авторизация пользователя. Еще один вариант дублирования кода — это когда разработчики создают собственную логику вместо того, чтобы использовать общепринятые утилиты. Хотя код при этом не копируется и не вставляется, результат остается тем же, что и при явном дублировании кода.

Мэри (разработчик). Вы не знаете, как мне перебрать коллекцию объектов `User`?

[6] JDepend — это инструмент анализа архитектуры и проекта исходного кода. Доступен на сайте *`www.clarkware.com/software/JDepend.html`*.

Адам (разработчик). На той неделе я написал код перебора. Вы можете найти его в пакете `User`.

Мэри. Прекрасно! Я скопирую его оттуда и использую. Спасибо.

Вот так и происходит дублирование кода. Если вы не знаете, где происходит дублирование и насколько часто, трудно определить, где и с какими проблемами вы столкнетесь при рефакторинге.

Решение

Чтобы принять решение, необходимо сначала оценить проблему. Для оповещения о дублировании исходного кода вы можете применять автоматизированные инструменты инспекции, такие как CPD от PMD[7] или инструмент статического анализа Simian8. Мы запускали их в составе процесса построения, поэтому могли использовать в любое время. Применяя эти инструменты, мы определяли области кода, которые имели большой процент дублирования, а затем обобщили код в компоненты. Использование данного подхода позволило непрерывно контролировать дублирование кода и уменьшать его объем в системе.

В типичном сценарии вы могли бы обнаружить, что несколько классов содержат тот же самый или подобный код. Чтобы уменьшить количество совпадающего кода, имеет смысл предпринять следующее.

1. Проанализируйте код, используя анализатор дублирования кода, такой как Simian или CPD от PMD. Включите его в сценарий построения.
2. В ходе рефакторинга[9] кода уменьшите количество повторяющегося кода, собрав его в единый метод или компонент, который вызывают те классы, где он используется.
3. Осуществляйте инспекции дублирования кода *непрерывно*, включив инспектор дублирования в систему CI. Это позволит вам выявлять сдублированный код сразу.

В главе 7, “Непрерывная инспекция”, подробно описано выполнение инспекции, их частота и способы применения.

Резюме

В этой главе обозначены ключевые области риска, снизить которые помогает CI, включая интеграцию базы данных, проверку, инспекцию, развертывание, обратную связь и документацию. Табл. 3.1 предоставляет краткий обзор материала, описанного в данной главе. Используя практики CI, вы можете снизить эти риски, улучшив качество программного обеспечения.

---

[7] CPD — утилита из метрических инструментов PMD, оповещающая об экземплярах скопированного и вставленного исходного кода. Доступна на сайте *http://pmd.sourceforge.net/*.

[8] Simian (Similarity Analyser — анализатор подобия) обеспечивает поддержку для языков C#, Java, Ruby и ряда других. Доступно для загрузки на сайте *www.redhillconsulting.com.au/products/simian/*.

[9] “Рефакторинг вносит изменения в код, улучшая его внутреннюю структуру, без воздействия на его внешнее поведение”. См *Refactoring with Martin Fowler: A Conversation with Martin Fowler, Part I* Билла Веннерса (Bill Venners) на `www.artima.com/intv/refactor.html`.

Таблица 3.1. Риски и их снижение

Риск	Снижение
Отсутствие развертываемого программного обеспечения	Используйте систему CI для построения развертываемого программного обеспечения в любое время. Организуйте воспроизводимый процесс построения, применяющий все элементы программного обеспечения из хранилища с контролем версий
Позднее выявление дефектов	Выполняйте построение, подразумевающее проверки разработчика при каждом изменении, это позволит обнаруживать дефекты программного обеспечения на раннем этапе разработки
Плохой контроль проекта	Выполняя построение регулярно, постоянно контролируйте состояние вашего программного обеспечения. При эффективном применении практик CI состояние проекта перестанет быть проблемой
Низкокачественное программное обеспечение	Запускайте проверки и инспекции при каждом изменении, так вы сможете обнаружить потенциальные дефекты базового кода, включая его чрезмерную сложность, дублирование, недостатки проекта, покрытие кода проверками и другие факторы

Вопросы

Насколько система CI позволяет снизить риски, подстерегающие ваш проект? Эти вопросы должны помочь вам выявить риски в проекте.

- Когда вы обнаруживаете больше дефектов в своем проекте, на начальных или на более поздних этапах разработки?
- Как вы определяете качество ваших проектов? Действительно ли вы способны измерить его?
- Какие процессы выполняются в ваших проектах вручную? Можете ли вы выявить процессы, подлежащие автоматизации?
- Имеете ли вы все сценарии для перепостроения вашей базы данных и данных в хранилище с контролем версий? Действительно ли вы способны перестроить вашу базу данных и проверочные данные в течение процесса построения?
- Действительно ли вы способны осуществлять регрессионную проверку после каждого изменения? Действительно ли вы способны выполнять различные типы регрессионных проверок, включая проверку функций, интеграции, загрузки и производительности?
- Можете ли вы определить, какой именно исходный код не имеет соответствующей проверки? Используете ли вы инструменты контроля покрытия проверками?
- Какой процент вашего программного обеспечения имеет сдвоенный код? Пытаетесь ли вы уменьшить его объем?
- Контролируете ли вы соответствие текущего исходного кода архитектуре программного обеспечения?
- Как вы оповещаете о готовности программного обеспечения к проверке? Используются ли в вашем проекте ручные механизмы связи, которые можно автоматизировать?
- Действительно ли вы способны просмотреть текущую схему вашего программного обеспечения? Как вы демонстрируете его архитектуру новому сотруднику проекта?

Глава 4

Построение программного обеспечения при каждом изменении

Вся проклятая Вселенная должна быть разобрана кирпич за кирпичом, а затем восстановлена.

Генри Миллер (Henry Miller), американский писатель и живописец (1891–1980)

В начале XX века рабочие на конвейере Форда собирали автомобили вручную. Сборка модели T занимала несколько дней. Нынешние автомобили во сто крат сложнее модели T, но их сборка происходит быстрее. Почему? Ответ прост: *автоматизация*. В автомобилестроении автоматизация освободила людей от выполнения повторяемых задач, поручив их роботам. Точно так же, используя автоматизированное построение, можно механизировать трудоемкие задачи процесса разработки программного обеспечения. Фактически в обеих отраслях промышленности прогресс был обусловлен ростом спроса. Когда рабочий монотонно трудится по восемь часов в день в основном руками, у него совершенно нет времени ни на усовершенствование продукта и процесса его производства, ни на планирование развития[1].

[1] А для чего тогда собственно инженеры? — *Примеч. ред.*

Иногда разработчики оказываются сапожниками без сапог, они создают приложения для автоматизации труда пользователей, но не автоматизируют собственные процессы *разработки* программного обеспечения. Анализ[2], проведенный в 2003 году, показал, что порядка 27 % групп разработки осуществляют ежедневное построение. По аналогии с автомобилестроением можно сказать, что мы все еще используем на конвейере старую добрую ручную сборку.

Зачастую люди оправдывают недостаточную автоматизацию разработки сложным характером программного обеспечения. Да, его разработка *бывает* сложной, но она имеет много повторяемых, склонных к ошибкам действий, которые вполне можно автоматизировать. Кроме того, несмотря на сложность разработки программного обеспечения, его *передача* может осуществляться нажатием одной кнопки.

Кнопка <Integrate> (как видно на рис. 4.1) представляет собой "автоматизированную линию сборки", которая воплощает множество практик, составляющих высокоуровневую практику CI. Автоматизированное построение представляет современную автоматизированную линию сборки, использующую "роботов" для интеграции программного обеспечения.

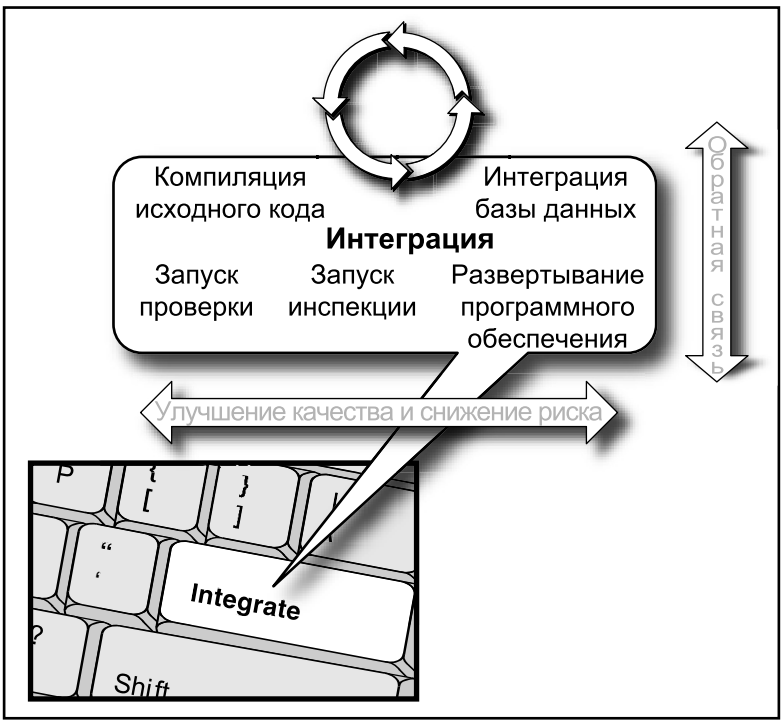


Рис. 4.1. Построение программного обеспечения способно улучшить его качество и снизить риск

[2] Цитата из "*Software Development Worldwide: The State of the Practice*" (авторы Алан Мак-Кормак (Alan MacCormack), Крис Кемерер (Chris Kemerer) и Билл Крендалл (Bill Crandall)), издательство *IEEE Software*, ноябрь-декабрь 2003, Т. 20, №6, с. 28–34. www.pitt.edu/~ckemerer/CK%20research%20papers/SwDevelopmentWorldwide_CusumanoMacCormackKemerer03.pdf.

В этой главе мы обсудим преимущества использования сервера CI для интеграционного построения при внесении любого изменения. Не все построения одинаковы, и впоследствии мы рассмотрим их типы и организацию. Мы также рассмотрим аспекты выбора и использования отдельной машины для интеграционного построения и CI. Автоматизация CI — это не единственный реальный подход реализации интеграционного построения; мы также ознакомимся с методикой выполнения интеграции вручную, с использованием поочередного подхода. Поскольку быстрая обратная связь при построении очень важна, мы закончим главу этой темой.

Автоматизируйте построения

Применяя автоматизированные сценарии построения, вы уменьшаете количество ручного труда в ведущих к ошибкам повторяемых процессах, выполняемых в проекте программного обеспечения.

Что такое *построение* (build) программного обеспечения? Только ли это *компиляция* (compiling) его компонентов? Или построение — это компиляция компонентов *и* запуск автоматизированных проверок? Должно ли построение включать *инспекции* (inspection)? Построение может включать любой из этих процессов, что эффективно снижает риски; однако чем больше процессов добавлено в построение, тем медленней обратная связь. Следовательно, вы должны решить, какие процессы следует включить в автоматизированное построение. Например, в главе 2, "Введение в непрерывную интеграцию", мы описали практику *закрытого построения* (private build), состоящего из интеграции изменений группы и *полного* построения (которое может включать компиляцию, проверку, инспекции и т.д.), выполняемого на вашей рабочей станции до передачи кода в хранилище с контролем версий, что предотвращает сбойные построения. С другой стороны, если вы хотите лишь опробовать пару изменений и не предполагаете передавать их, вы можете осуществить облегченное построение, т.е. выполнить только компиляцию и несколько проверок модуля.

"Ant великий?"

Большинство примеров, приведенных в этой книге, используют инструменты построения Ant и NAnt. Это связано с их особой популярностью среди разработчиков. Я ожидаю (и надеюсь), что новые инструменты построения, поддерживающие зависимости *и* программирование, станут в последующие годы более широко распространенными.

Существует множество инструментов построения. К наиболее популярным относятся Ant для Java и NAnt для .NET. Использование инструментов выполнения сценариев, разработанных специально для построения программного обеспечения, вместо собственного набора пакетных файлов оболочки, является более эффективным способом создания однозначного и воспроизводимого решения построения.

Помните: построение должно осуществляться нажатием одной кнопки. Когда вы нажимаете кнопку <Integrate>, как показано на рис. 4.1, на сборочном конвейере запускается процесс, который создает *работоспособное* программное обеспечение. Иногда организации не способны задействовать CI потому, что они не могут *реально* автоматизировать свое построение. В некоторых случаях такая неспособность обусловлена жесткой связью и зависимостями, например с библиотеками сторонних производителей и жестко заданными ссылками. Я не раз видел проекты с яркими примерами следующего.

- Жесткая зависимость от совместно используемых дисков. Например, сценарии построения жестко привязаны к диску `K:\` (когда на машине разработчика диск "K" недоступен, возникает проблема).
- Существование жестко заданных ссылок на положение (диск `C:\`) некоторых инструментов, отсутствующих на машине разработчика.

В обоих случаях сценарии становятся неработоспособными не только на машине с операционной системой, отличной от Windows, они могут также не сработать на машине разработчика, если он иначе подключит сетевой диск или переместит инструменты с диска `C:\` в другой каталог. Попытка запуска подобного сценария при невозможности найти зависимый элемент приводит к сбою и неудаче построения.

Случалось ли вам видеть программное обеспечение, которое, не проходя проверок, оказалось работоспособным? А как насчет программного обеспечения, которое было проверено, но не проинспектировано? Предположим, некто вам скажет, что "здесь все работает, кроме базы данных" — это что, работоспособное программное обеспечение? Некоторые разработчики полагают, что их программное обеспечение функционирует, если оно компилируется. Существуют разные типы построения (рассматриваемые далее в этой главе), поэтому вы будете балансировать между тяжеловесным построением, включающем все проверки, но производящем рабочее, развертываемое программное обеспечение (как правило, прошедшее многие типы проверок, а также инспекций), и необходимостью получения быстрой обратной связи.

Выполняйте построение одной командой

Мартин Фаулер советует: "Получайте все необходимое из хранилища с контролем версий, чтобы можно было построить целую систему, отдав одну команду"[3]. Концепция кнопки <Integrate> реализуема только тогда, когда вы можете запускать построение одной командой. Например, ввод в командной строке команды `nant integrate`, как представлено в листинге 4.1, является примером единой команды, инициализирующей интеграционное построение.

Листинг 4.1. Сценарий построения, запускаемый одной командой

```
> nant integrate
Buildfile: file:///C:/dev/projects/acme/project.build
clean:
svn-update:
all:
compile-src:
compile-tests:
integrate-database:
run-tests:
run-inspections:
package:
deploy:
BUILD SUCCEEDED
Total time: 3 minutes 13 seconds
```

[3] См. *Continuous Integration* на `www.martinfowler.com/articles/continuousIntegration.html`.

Для работы в автоматизированном режиме сервер CI нуждается в “безголовом” процессе, например в едином сценарии команды. При выполнении интеграционного построения на отдельной машине нельзя полагаться на IDE. Кроме того, для осуществления интеграционного построения единой командой необходимо получить доступ ко всем элементам программного обеспечения (из хранилища с контролем версий).

Автоматизированное построение — это как кнопка <Integrate>: нажмите кнопку, и ваше программное обеспечение будет построено (и развернуто). Это означает, что все элементы последнего будут связны, а их функционирование проверено. Рис. 4.2 иллюстрирует действия, обычно выполняемые сценарием построения.

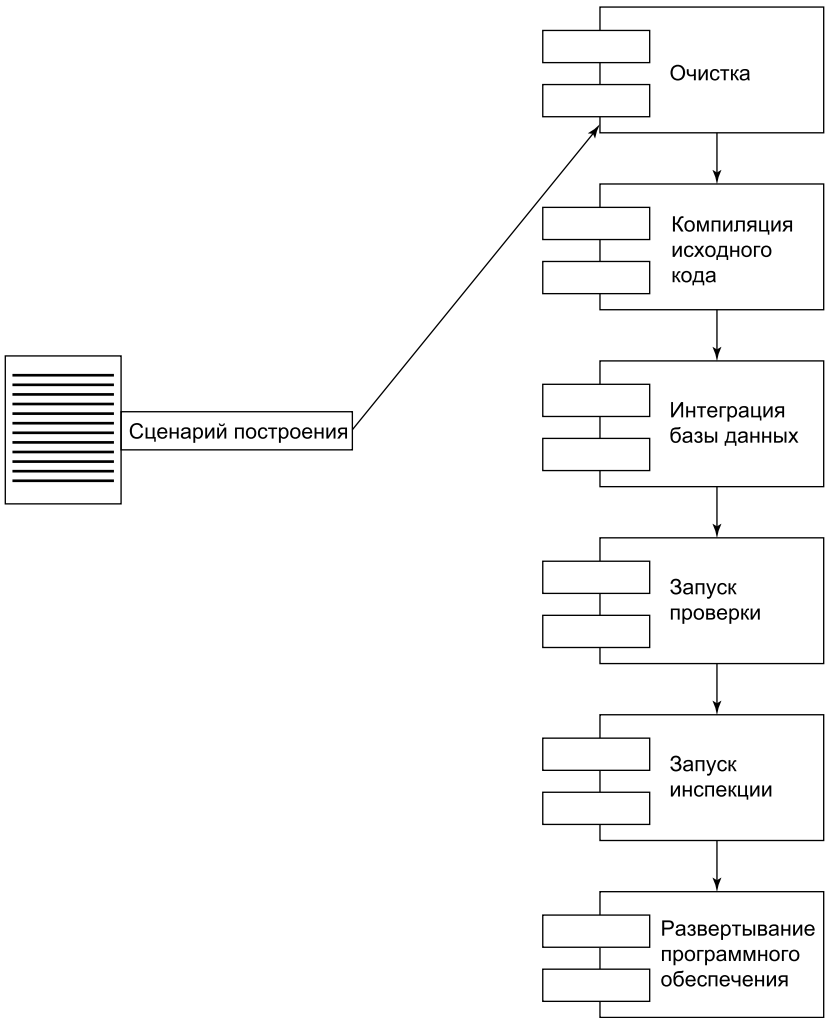


Рис. 4.2. Логические процессы сценария построения

В общих чертах построение программного обеспечения подразумевает примерно следующие этапы.

1. Осуществляйте построение с использованием таких инструментов сценариев построения, как NAnt, Rake, Ant или Maven. Сначала оставьте сценарии простыми; впоследствии в них можно будет добавить больше процессов.
2. Добавьте кнопке <Integrate> внутри сценария построения другие процессы (очистка, компиляция и т.д.).
3. Чтобы построить программное обеспечение, запустите сценарий из IDE или командной строки.

Листинги 4.2 – 4.6 демонстрируют примеры использования инструмента построения NAnt для платформы .NET; но вы вполне можете получить тот же результат при помощи других средств сценариев построения, таких как Ant или Maven для Java, MSBuild для .NET, Rake для Ruby и т.д. Проблема не в том, какой инструмент вы выбираете, главное, использовать существующий инструмент, а не создавать собственное решение.

Листинг 4.2 демонстрирует сценарий NAnt, использующий задачу `delete` для удаления всех каталогов и файлов перед новым построением. Это уменьшает вероятность того, что файлы предыдущего построения неблагоприятно повлияют на новое построение.

Листинг 4.2. Очистка созданных каталогов с использованием NAnt

```
<target name="clean">
    <delete dir="${build.dir}" verbose="true" failonerror="false"/>
    <delete dir="${dist.dir}" verbose="true" failonerror="false"/>
    <delete dir="${reports.dir}" verbose="true" failonerror="false"/>
</target>
```

Листинг 4.3 демонстрирует компиляцию кода C# с использованием задачи `csc`, которая компилирует все файлы в некотором каталоге и перемещает полученный файл `.dll` в другой каталог. Вторая часть данного примера демонстрирует запуск сценария SQL, выполняющего определения данных и создающего таблицы в базе данных.

Листинг 4.3. Компиляция и перепостроение базы данных с использованием NAnt

```
<target name="build">
    <csc target="library" debug="${build.debug}"
        output="${build.dir}\bin\${config}\${nant.project.name}.dll">
            <sources failonempty="true">
                <include name="${project.localpath}/**/*.cs" />
            </sources>
    </csc>
</target>
<target name="integrate-database">
    <sql connstring="${project.db.conn}"
        delimiter=";"
        delimstyle="Normal"
        print="true"
        source="${data-definitions}"/>
</target>
```

Листинг 4.4 содержит пример выполнения задачи `nunit2` в NAnt, выполняющей комплекс проверок NUnit. Обратите внимание, что при сбое любой из проверок происходит общий сбой построения (как можно заметить, атрибуту `failonerror` задачи `nunit2` присвоено значение `true`). Как упоминалось в главе 2, “Введение в непрерывную интеграцию”, для корректности построения *все проверки и инспекции должны быть пройдены.*

Листинг 4.4. Проверка с использованием NUnit и NAnt

```
<target name="run-tests" depends="compile-src">
    <nunit2 failonerror="true">
        <formatter type="Xml"
              usefile="true"
              extension=".xml"
              outputdir="${build.dir}/results"/>
            <test assemblyname="${build.dir}\bin\${config}
↳\${project}.Test.dll"
             appconfig="mydefaulttest.config"/>
    </nunit2>
</target>
```

Листинг 4.5 демонстрирует выполнение задачи `fxcop`, запускающей FxCop, бесплатный инструмент для платформы .NET, который инспектирует и оповещает о предопределенных нарушениях кода, связанных с производительностью, защитой, соглашениями именования и т.д.

Листинг 4.5. Инспекция с использованием FxCop и NAnt

```
<target name="fxcop">
    <fxcop>
        <targets>
            <include
                name="${build.dir}\bin\${config}\${project}.dll"/>
        </targets>
        <arg value="/out:${build.dir}\bin\${config}\fxcop.xml"/>
    </fxcop>
</target>
```

Последнее действие построения, представленное на рис. 4.2, — развертывание. Листинг 4.6 иллюстрирует использование задачи NAnt для простого развертывания на сервере FTP.

Листинг 4.6. Развертывание с использованием FTP и NAnt

```
<target name="deploy">
    <connection id="staging"
        server="devqa.ib.com"
        username="helloworld"
        password="myftppwd" />
    <ftp connection="staging"
        remotedir="incoming"
        localdir="c:\dev\project\acme">
        <put type="bin">
            <include
              name="${build.dir}\bin\${config}\${project}.dll" />
        </put>
    </ftp>
</target>
```

Если сценарий построения выполнен разработчиком безо всякой обратной связи, то он даже не будет знать, прошло ли построение успешно. Очень простой пример уведомления об отказе включен в листинг 4.4. Если любая из проверок, запущенных задачей `nunit2`,

закончится неудачей, то *все* построение будет считаться сбойным. Фактически NAnt прекратит работу с сообщением `BUILD FAILED`, так что никаких сомнений не будет.

Это ни в коем случае не исчерпывающий пример сценария построения. Сценарий построения, который полностью имитирует кнопку <Integrate>, должен был бы включать намного больше процессов.[4]

Отделяйте сценарии построения от IDE

Вы должны избегать применения IDE в сценариях построения. IDE может зависеть от сценария построения, но *сценарий построения не должен зависеть от IDE*. Рис. 4.3 иллюстрирует соответствующую зависимость. Она иногда более тонкая, чем вы можете думать. Например, IDE способен облегчить создание сценария построения, но при этом он может помещать файлы построения и зависимые элементы в структуре того каталога, в котором установлен IDE. Чтобы проверить возможность многократного использования сценария построения, созданного IDE, возьмите сценарий построения и запустите его на новой машине с только что установленной операционной системой (и соответствующим инструментом построения).

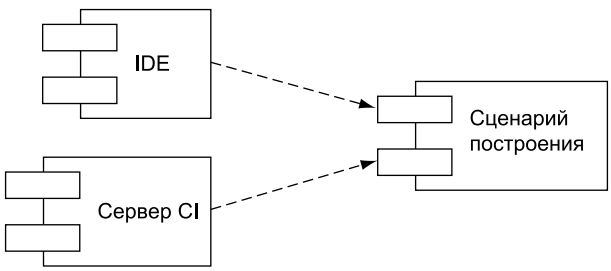


Рис. 4.3. Отсоединение сценария построения от IDE

Создание отдельного сценария построения важно по двум причинам.

1. Все разработчики могут использовать разные IDE, поэтому может оказаться весьма сложно соотнести конфигурационные особенности каждого IDE.
2. Сервер CI должен работать автоматически, осуществляя построение без вмешательства людей. Следовательно, тот же автоматизированный сценарий построения, используемый разработчиками, может и должен быть использован сервером CI.

Централизуйте элементы программного обеспечения

Для эффективного построения программного обеспечения все его элементы следует централизовать.

Централизуя элементы программного обеспечения в системе с контролем версий, вы способствуете построению единой командой, описанному ранее в этой главе. Кроме того, централизация помогает предотвратить проблему “но на моей машине это работает”, ког-

[4] Более подробная информация по этой теме приведена на поддерживающем книгу Web-сайте *www.integratebutton.com/*.

да разработчик не способен воспроизвести дефект, который произошел в некой другой системе, например на проверочной машине или машине пользователя. В этом разделе мы рассмотрим различные методы централизации элементов программного обеспечения.

Один из подходов централизации элементов программного обеспечения подразумевает использование хранилища с контролем версий для хранения всех файлов. В книге Стивена Беркзука (Stephen Berczuk) и Бреда Апплетона (Brad Appleton) *Software Configuration Management Patterns* это называется "Repository pattern" (*схема хранилища*) и подразумевает, что "рабочее пространство состоит не только из кода", а включает следующее:

- компоненты в виде исходных файлов или библиотек;
- компоненты стороннего производителя, такие как файлы JAR, библиотеки, файлы DLL5 и другие файлы, в зависимости от языка и платформы;
- файлы конфигурации;
- файлы данных для инициализации приложения;
- сценарии построения и параметры среды построения;
- сценарии установки для некоторых компонентов[6].

При использовании хранилища с контролем версий для централизации *всех* элементов программного обеспечения остается определить, что же именно составляет это "все". Можно использовать уровни или рискнуть и решиться на минимум типов элементов программного обеспечения, которые будут находиться в хранилище с контролем версий. Например, одним из рисков для продукта с большой продолжительностью существования является то, что последующие версии компиляторов и инструментов могут столкнуться с проблемами, возможно незначительными и незаметными. Это риск обусловлен тем, что вам может понадобиться перекомпилировать прежнюю версию.

Кроме того, некоторые версии инструментальных средств плохо совмещаются с другими. Разработчику достаточно просто задействовать любую версию инструмента, которую он считает подходящей, однако гоняясь за новыми преимуществами, можно обнаружить и новые недостатки. Аналогично, возвращение и попытка воссоздания прежнего построения (например, для воспроизведения проблемы клиента или устранения ошибки) может потребовать определенного набора инструментальных средств, которые использовались при разработке в *прежнее время*. Таким образом, приходим к заключению, что, вероятно, практически нет никаких элементов вашего проекта, которые не пригодились бы в будущем по разным причинам. В этом и проявляется ценность для проекта централизованного хранилища элементов с контролем версий.

Создайте строгую структуру каталога

Использование хранилища с контролем версий для управления всеми элементами программного обеспечения может остаться лишь разговорами, поскольку для его реального воплощения сценарии должны уметь находить все необходимое на сервере CI. Чтобы обеспечить реальную возможность находить и получать из хранилища все бесчисленные комбинации элементов, которые могут быть использованы в проекте, вам придется создать однозначную, логичную структуру каталога.

---

[5] Собственно файлы JAR и DLL — это и есть библиотеки. — *Примеч. ред.*

[6] Из книги Стивена Беркзука (Stephen Berczuk) и Бреда Апплетона (Brad Appleton) *Software Configuration Management Patterns.*

Один из подходов подразумевает создание структуры каталога на типичных действиях рабочего проекта, например требования, проектирование, реализация и проверка. Используете ли вы такую структуру или другую, главное, чтобы она хранила различия в содержимом понятно и однозначно. Кроме того, очень важно, чтобы каждая задача в построении выполнялась из каталога, который содержит *только исходный код и сценарии, связанные с этой задачей*, а не весь проект. Например, ваш сценарий интеграционного построения может получать весь исходный код и связанные сценарии из каталога `implementation`. Это может значительно ускорить построение, поскольку поиск всех необходимых файлов (документов и двоичных файлов) может проходить довольно долго. Простая структура каталога, подобная приведенной ниже, поможет отделить файлы исходного кода от других файлов, упростив запуск построения.

- `implementation`
- `requirements`
- `design`
- `management`
- `deployment`
- `testing`
- `tools`

Безусловно, все каталоги верхнего уровня могут содержать множество вложенных каталогов. Каталог `implementation` должен содержать только файлы исходного кода и может выступать в роли первичного каталога для построения.

Ранний сбой построения

В хорошо организованном построении неудача проявляется быстро. Весьма огорчает сбой после успешного выполнения множества других элементов построения, когда уже потрачено драгоценное время. Чтобы сбой при построении происходил как можно раньше, нужно предпринять следующие действия.

1. Интегрируйте компоненты (получите из хранилища последние изменения и откомпилируйте).
2. Запустите реальные проверки модуля (т.е. быстрые проверки, которые не затрагивают базу данных и любые другие зависимые элементы).
3. Запустите другие автоматизированные процессы (перепостроение базы данных, инспекция и развертывание).

Это лишь одна из рекомендуемых последовательностей построения. Все зависит от того, где наиболее вероятен сбой в специфическом проекте. Чем больше вероятность сбоя некоего процесса, тем раньше его следует запустить в сценарии построения. Имейте также в виду, что порядок выполнения иногда диктуется последовательностью построения. Например, компиляция исходного кода происходит перед его проверкой. Эффективность построения выше, если склонный к отказу процесс выполняется раньше. Раздел “Выполняйте быстрое построение” далее в этой главе описывает способы организации построения, позволяющие уменьшить его продолжительность и обеспечить более быструю обратную связь.

Осуществляйте построение для каждой среды

Зачастую проект программного обеспечения требует развертывания в различных средах. Для этого вам может пригодиться наличие в хранилище различных файлов конфигурации, предназначенных для настройки разных систем (разработка, интеграция, проверка, QA и работа), включая файлы `.properties`, `.xml` или `.ini`. Каждая платформа, язык и инструмент сценариев будет иметь собственный вариант конфигурации. Перенастройка построения основана на его сценарии, который выбирает предопределенные конфигурации, доступные в программном обеспечении, не изменяя базовые функции сценариев построения. В большинстве случаев вы можете снабдить конфигурацию "ловушками" заменой файлов конфигурации, используемых приложением. Можно также настроить среды выполнения и API, с которыми взаимодействует приложение. Конкретная методика зависит от платформы и принятых соглашений. Ниже приведен список перестраиваемых значений конфигурации, присущих большинству сред.

- Достоверность регистрации.
- Настройка сервера приложений.
- Информация для подключения к базе данных.
- Конфигурация среды выполнения.

Хотя среды, в которых осуществляются проверки или развертывание, могут разниться, сценарии построения не должны быть разными. Файлы конфигурации (типа `.properties` или `.include`) позволяют вам перебирать варианты без необходимости копировать и вставлять значения, подходящие для каждой среды внутри сценариев построения. Это еще одна область, где, как и в исходном коде, дублирование ведет к большим проблемам и снижению надежности. Для гарантии возможности создания работоспособного программного обеспечения в любой среде следует улучшить способность сценария построения к настройке за счет применения параметров. Как показано на рис. 4.4, вы можете запускать тот же самый сценарий построения и предоставить соответствующий файл свойств, настраивающий построение для каждой среды. Например, при развертывании в среде QA вы можете вызывать сценарий построения так:

```
ant -f build.xml -D environment=qa
```[7]

Здесь `environment` — свойство, которое было определено в сценарии Ant. Параметр `-D` означает, что это системный параметр, передаваемый сценарию Ant.

Типы и механизмы построения

Существуют разные типы построения, предназначенные для различных целей. Построение может быть запущено разными механизмами, например пользователем, по расписанию, в связи с внесением изменений или в результате некоего события.

Типы построения

В иерархии типов построения три уровня: для отдельных лиц, групп и пользователей, т.е. закрытое построение, интеграционное построение (объединяющее результат работы с работой остальной части группы) и финальное построение, в результате которого получается программное обеспечение, передаваемое пользователям.

[7] В первоисточнике пробела между ***-D*** и ***environment*** нет. — *Примеч. ред.*

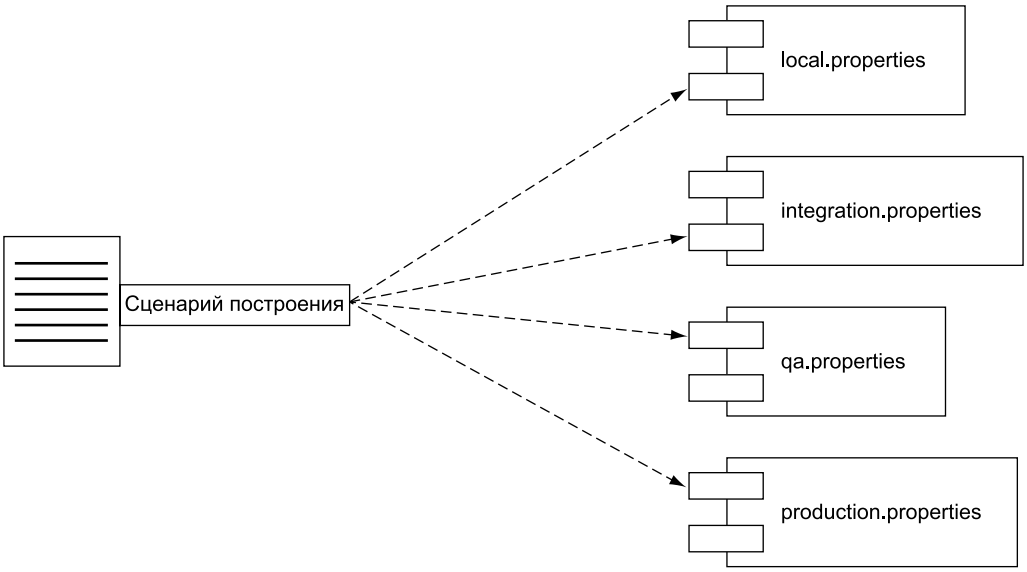


Рис. 4.4. Настраиваемое построение для разных сред

Закрытое построение

Разработчик выполняет *закрытое построение* (private build) перед передачей своего кода в хранилище. Осуществляя закрытое построение, вы интегрируете свои изменения с последними изменениями, доступными в хранилище с контролем версий. Это может предотвратить сбой построения. При закрытом построении выполняются следующие шаги.

1. Проверьте код, полученный из хранилища.
2. Внесите в него изменения.
3. Получите последние системные изменения из хранилища.
4. Запустите построение, включающее выполнение всех ваших проверок модуля.
5. Передайте свои изменения кода в хранилище.

Интеграционное построение

Интеграционное построение (integration build) интегрирует изменения, внесенные в хранилище группой с *общей линией* (mainline) (называемой также *головой* (head) или *магистралью* (trunk)). В идеале оно должно осуществляться на выделенной машине.

Фаулер[8] рассматривает различные типы построения, которые могут быть выполнены в ходе интеграционного построения. Он называет это "*поэтапным построением*" ("staged builds"), которое включает "*передающее построение*" ("commit build") и "*последующие построения*" ("secondary builds"). Передающее построение — это ваше самое быстрое интеграционное построение (меньше десяти минут), включающее компиляцию и проверку модуля. Последующее построение — это интеграционное построение, которое запускает более медленные проверки, например проверку компонентов, системы и эксплуатационные

[8] См. *Continuous Integration* на `www.martinfowler.com/articles/continuousIntegration.html`.

испытания. Здесь могут также осуществляться автоматизированные инспекции соблюдения стандартов программирования и сложности кода.

Финальное построение

Финальное построение (release build) готовит программное обеспечение к выпуску для пользователей. Одной из задач CI является создание развертываемого программного обеспечения. Финальное построение, происходящее в конце итерации или некоторого другого промежуточного этапа, может включать более обширные проверки, в том числе проверку производительности, загруженности и все приемочные испытания. Кроме того, при большинстве финальных построений создается установочная среда, запускаемая в системе пользователя. Финальное построение может также применяться для проверки готовности к QA, если используется отдельный, поэтапный процесс и группа.

Механизмы построения

Не все построения запускаются одинаково. Для вызова построения в соответствующий момент необходимо учесть его назначение и частоту. В некоторых ситуациях сценарии могут оказаться настолько большими или иметь так много зависимостей, что запускать их автоматически не стоит; лучше делать это по требованию. В других случаях автоматический запуск может выполняться под управлением CI. Типы механизмов построения описаны ниже.

- *По требованию* (on-demand). Это управляемый пользователем процесс, в котором некто вручную инициализирует интеграционное построение.
- *По расписанию* (scheduled). Процессы управляются по времени, например, можно ежечасно проверять, не произошли ли изменения. Расписание действий может пригодиться для повторяемых процессов, таких как запуск исчерпывающего набора проверок защиты или загруженности программного обеспечения. Для расписания можно использовать задачу `cron`, хотя большинство серверов CI самостоятельно поддерживают расписание.
- *Опрос изменений* (poll for changes). Регулярно инициируемый процесс проверки хранилища с контролем версий на предмет изменений. Если таковые обнаружены, он запускает интеграционное построение. Все серверы CI поддерживают некий механизм “опроса изменений”.
- *Управляемое событиями* (event-driven). Управление событиями подобно опросу изменений, но вместо инструмента CI построение запускает хранилище с контролем версий на основании предопределенного события (изменения). Если хранилище с контролем версий обнаруживает изменение, оно инициализирует сценарий построения.

Запуск построения

Табл. 4.1 демонстрирует взаимосвязь между типом построения и способом его запуска.

Таблица 4.1. Запуск построения с использованием различных механизмов

Тип построения	Механизм построения
Закрытое	По требованию
Интеграционное	По требованию, опрос изменений, по расписанию, управляемое событиями
Финальное	По требованию, по расписанию

Используйте выделенную машину для интеграционного построения

Когда вы выделяете машину для интеграционного построения, вы решительно ограничиваете предположения о среде и конфигурации, а также способствуете предотвращению слишком позднего проявления проблемы "а на моей машине это работает". Любая локальная рабочая станция обычно имеет несколько разных конфигураций и множество зависимостей, зачастую отсутствующих в среде развертывания. Если разработчик вносит локальные изменения и забывает передать несколько файлов в хранилище с контролем версий, то система CI, выполняющаяся на отдельной машине, запустит интеграционное построение и обнаружит их отсутствие. Кроме того, вы можете устанавливать серверы приложений и баз данных в определенное состояние каждый раз, когда происходит интеграционное построение. Это также позволит не только уменьшить количество предположений, но и существенно быстрей обнаруживать и решать проблемы. Когда сотрудники узнают, что последнее интеграционное построение потерпело неудачу, они могут избежать получения сбойного исходного кода из хранилища с контролем версий. Машина интеграционного построения действует как сеть безопасности, гарантируя, что программное обеспечение работает как ожидалось.

Многих зачастую интересует, во сколько обойдется приобретение выделенной машины для интеграционного построения. Это важный вопрос, имеющий даже более важный ответ. Приведенный ниже диалог демонстрирует, как быстро этот вопрос теряет значение.

Питер (технический руководитель). Я хотел бы приобрести для нашего проекта Logistics выделенную машину интеграционного построения.

Билл (руководитель проекта). Зачем вам отдельная машина?

Питер. Чтобы мы могли строить наше программное обеспечение непосредственно из хранилища Subversion при каждом изменении. Мы также будем иметь возможность переустанавливать среду, включая проверочные данные. Все это позволит быстрее находить и устранять проблемы.

Билл. Звучит заманчиво, но, Питер, у нас действительно нет денег на это. Я полагаю, что вам потребуется по крайней мере 1 000 долларов, не так ли?

Питер. Кстати, причина, по которой мы провели последнюю субботу на работе, была связана с проблемой интеграции. Получив машину интеграционного построения, мы сможем значительно сэкономить время и деньги. Сэкономленное время с лихвой окупит стоимость машины, причем много раз. В понедельник мы были вынуждены вручную интегрировать и проверять демонстрационную версию приложения. Нам действительно нужна эта машина, чтобы автоматически интегрировать программное обеспечение при каждом изменении.

Билл. Ладно, но все, что мы можем сделать сейчас, это задействовать одну из дополнительных машин в серверной. Вы можете удалить с нее все и сделать машиной интеграционного построения.

Судя по диалогу Билла и Питера, вам, возможно, и не придется тратить деньги на приобретение новой машины. Машиной CI может стать любая неиспользуемая дополнительная машина. Если выделенная машина интеграционного построения недоступна, используйте для начала собственную машину разработки. Это лучше, чем отказ от интеграции вообще, но это не долгосрочное решение. Убедитесь, что использовали отдельное место на своей машине (т.е. каталог или раздел).

При создании машины интеграционного построения следует учитывать несколько факторов. Сосредоточившись на них, вы извлечете максимум преимуществ.

- *Рекомендуемые системные ресурсы.* Многие из них можно получить при помощи инструментальных средств. Чем лучше аппаратные ресурсы, тем меньше продолжительность построения (обсуждается далее в этой главе). Как правило, цена увеличения аппаратных ресурсов машины интеграционного построения окупается за счет экономии времени.
- В хранилище с контролем версий находятся все элементы программного обеспечения. Все, что имеет отношение к его разработке, должно быть передано в хранилище с контролем версий. Сюда относится исходный код, сценарии построения, файлы конфигурации, инструменты (сервер приложений, сервер баз данных и инструменты статического анализа), сценарии проверки кода и файлы базы данных (см. раздел "Централизуйте элементы программного обеспечения" ранее в этой главе).
- *Чистая среда.* Перед выполнением интеграционного построения сценарий CI должен удалять любые зависимости кода от среды интеграции. Следует гарантировать удаление всего исходного кода и бинарных файлов предыдущего интеграционного построения. Удостоверитесь также, что система CI устанавливает проверочные данные и любые другие элементы конфигурации в определенное состояние. Этот подход уменьшает количество предположений, ликвидирует зависимости и организует построение программного обеспечения как будто на новой машине.

Наличие выделенной машины построения, которая способна его эффективно выполнять, позволяет запускать его часто. Кроме того, среда построения становится четко воспроизводимой, уменьшая количество предположений, относящихся именно к ней.

Волшебная машина

Большинство разработчиков рано или поздно сталкиваются с этой машиной[9]. Типичная ситуация: вы написали и полностью проверили программное обеспечение, но при развертывании его на другой машине (например, на проверочной) что-то не работает. Причин может быть множество: возможно, вы забыли передать файл в хранилище с контролем версий или проверочная машина настроена иначе, или механизм объединения сервера приложения рассчитан на меньшее количество подключений. В любом случае причина кроется в каком-либо *различии* между вашей и другой машиной (машинами). Это тот случай, когда вы восклицаете: "Но на моей машине это работает!" — поскольку на своей машине вы даже не можете получить эту проблему. Возможно, у вас "волшебная" машина?

"*Волшебные машины* (magic machine) — это одна из разновидностей волшебных аппаратных средств, которые оказываются единственными машинами компании, способными к построению программного обеспечения. Этот случай не столь фантастичен, как это может показаться. На протяжении своей карьеры я сталкивался с такими магическими чудовищами неоднократно. На первый взгляд машина одержима демоном, однако после ликвидации зависимостей и удаления бинарного мусора это проходит. Вот как обычная машина в инфраструктуре компании может превратиться в заколдованную: как-то раз разработчик по неосторожности создает жесткую зависимость со сценарием машины, например ссылку с полностью определенным путем каталога, или даже устанавливает некий инструмент, который существует только на его машине. Так он нечаянно предотвращает возможность построения и запуска приложения на любой другой машине."[10]

[9] Святая правда. — *Примеч. ред.*

[10] См. `http://www-128.ibm.com/developerworks/java/library/j-ap10106/index.html`.

Волшебные машины возникают и из-за "аппаратных" зависимостей от машины, на которой осуществляется построение. Это может случиться и на машине интеграционного построения, иногда на ней добавляют переменную среды окружения или изменяют конфигурацию, чтобы решить некую проблему, но забывают написать сценарий, применяющий то же на другой машине. Если бы ваша машина построения отказала и вы оказались не в состоянии работать, каково это было бы? И сколько бы времени заняла ликвидация проблемы и перезапуск машины?

Существует много решений данной проблемы. Вы можете включить в сценарий построения большинство ваших зависимостей и расположить их в хранилище. Можно создать макет некоторых зависимостей, например базы данных или сервера приложений. Кроме того, вы можете осуществить рефакторинг своих сценариев построения, чтобы удалить большинство жестко заданных зависимостей (например, переменные окружения) и заменить их относительными ссылками.

Используйте сервер CI

При реализации непрерывной интеграции имеет смысл использовать сервер CI. Безусловно, вы можете создать свой собственный инструмент или выполнять интеграцию вручную; но сейчас на рынке доступно множество превосходных инструментов, которые предоставляют ценнейшие возможности, а также позволяют расширять их. Следовательно, необходимость в создании собственного сервера CI отпадает. Но если вам все же придется делать это, то вы, вероятно, захотели бы включить в него большинство следующих возможностей.

- Периодический опрос хранилища с контролем версий на предмет изменений.
- Выполнение неких действий по расписанию, например ежечасно или ежедневно.
- Выявление "периодов затишья", в течение которых никаких интеграционных построений не выполняется.
- Поддержку для различных инструментов сценариев построения, включая утилиты командной строки, такие как Rake, make, Ant или NAnt.
- Отправку электронной почты заинтересованным сторонам.
- Отображение истории предыдущего построения.
- Отображение панели управления, доступной через Web, чтобы каждый мог просматривать информацию интеграционного построения.
- Поддержку нескольких систем контроля версий для разных проектов.

На этом список не заканчивается. Большинство серверов CI уже реализовало эти средства. Уверен, что можно без проблем подобрать инструмент, который полностью удовлетворяет вашим потребностям и подходит к среде разработки. CruiseControl, Luntbuild, Continuum, Pulse, и Gauntlet — вот лишь некоторые из инструментов, которые можно использовать для реализации CI. В приложении Б, "Обсуждение инструментальных средств CI", рассматриваются и оцениваются различные инструменты CI, присутствующие на рынке на момент написания этой книги.

Приходилось ли вам использовать сервер CI, интеграцию вручную или их комбинацию? Это вопрос риторический. Мы безусловно одобряем применение сервера CI. Но иногда возникают резонные основания для выполнения интеграции вручную, особенно с учетом минимальной инструментальной поддержки для предотвращения передачи сбойного кода в хранилище.

Выполняйте интеграционное построение вручную

Альтернативная или дополнительная методика использования сервера CI заключается в выполнении интеграции вручную. Практика ручного интеграционного построения подразумевает, что только один человек одновременно может передавать изменения в хранилище. Разработчики по очереди вручную запускают интеграционное построение на отдельной машине, гарантируя, что построение всегда остается в "зеленом" состоянии.

Используя сервер CI, ваша группа осуществляет *автоматизированное* (непоследовательное) интеграционное построение, которое может быть запущено в любое время. Как уже упоминалось в книге ранее, сервер CI может опрашивать хранилище на предмет изменений каждые несколько минут или около того. Как только он обнаруживает изменения, он запускает автоматизированное построение. Проблема автоматизированной интеграции заключается в том, что ее результатом может оказаться сбойное построение. В такой ситуации CI скорее недостаток. Проблема зачастую не обнаруживается до тех пор, пока ее причина не окажется в хранилище, а это означает, что другие разработчики могут успеть получить из хранилища сбойный код. Кроме того, сбойное построение может нарушить процесс работы разработчиков.

Чтобы предотвратить попадание сбойного кода в хранилище, некоторые группы используют последовательную интеграцию вручную или применяют физический маркер (я знал одну группу, которая использовала кнопку скобок) или обычную блокировку файла, чтобы обозначить только одного разработчика (или пару), который осуществляет интеграцию в данный момент.

Ручная интеграция может быть эффективна для предотвращения сбойных построений, но для больших групп она подходит не особенно хорошо. Кроме того, побочный эффект этой формы интеграции заключается в том, что участники группы могут накопить в очереди много изменений, а поскольку интеграции проходят реже, они оказываются крупней и сложней, что затрудняет получение качественного развертываемого программного обеспечения. К тому же при осуществлении интеграции исключительно вручную нет никакой гарантии, что все практики будет соблюдены. Использование сервера CI обеспечивает защищенную сеть, оповещающую о ходе интеграции. Некоторые группы применяют комбинацию последовательных и автоматизированных интеграций. Например, группа, которая использует исключительно автоматизированную CI, может комбинировать данный подход с отдельными закрытыми построениями каждым разработчиком, чтобы предотвратить нарушение интеграционных построений. Однако большинство из этого — вопрос личных предпочтений. Мы, безусловно, склоняемся к автоматизированной интеграции, но не отрицаем уникальных преимуществ ручной и последовательной интеграций, позволяющих поддерживать построение в "зеленом" состоянии.

Выполняйте быстрое построение

Остановка работы на время ожидания обратной связи построения замедляет ритм разработки проекта. Следовательно, если построение занимает слишком много времени, это отрицательно сказывается на практике CI. Быстрая обратная связь является определяющим фактором CI. Чем короче продолжительность интеграционного построения, тем быстрее вы получите обратную связь.

Масштабируемость и производительность интеграционного построения

Масштабируемость построения (build scalability) определяет, насколько ваша система построения пригодна к увеличению количества интегрируемого и анализируемого кода. *Производительность построения* (build performance) определяет его продолжительность. В идеале, по мере увеличения объема базового кода ваша система CI должна оставаться способной обрабатывать его без существенного снижения производительности.

Причиной медленного интеграционного построения может быть редкая передача кода разработчиками в хранилище с контролем версий. Начинать уменьшение продолжительности построения имеет смысл с общего анализа системы интеграционного построения на предмет выявления узких мест. Затем проанализируйте результаты и, определив наиболее подходящие средства, попробуйте внести изменения в процесс построения, возможно, они уменьшат его продолжительность. Наконец, еще раз оцените продолжительность построения и определите, стоит ли принимать дальнейшие меры.

В любом случае вы можете использовать следующую последовательность действий для диагностики и уменьшения продолжительности построения.

1. Соберите показатели построения.
2. Проанализируйте их.
3. Выберите и осуществите усовершенствования.
4. Снова оцените построение и при необходимости продолжите улучшения.

Десятиминутное построение

Во втором издании книги *Extreme Programming Explained* Кент Бек (Kent Beck) приводит хорошее эмпирическое правило, согласно которому построение (интеграция) должно занимать не более десяти минут. Большинство разработчиков, использующих CI, не переходят к следующей задаче, пока их последняя интеграция не увенчается успехом. Следовательно, построение, занимающее больше десяти минут, может существенно замедлить их работу. Данная рекомендация может подойти для большинства проектов. Ваше десятиминутное передающее построение не должно включать все типы проверок и инспекций. Вы можете сэкономить время, последовательно применяя разные типы построения (как уже упоминалось, Фаулер называет это “поэтапным построением”).

Сбор показателей построения

Сбор показателей построения — это первый шаг к его ускорению. Табл. 4.2 содержит список наиболее общих показателей, которые можно использовать при качественном анализе процесса интеграционного построения. Собирать все эти показатели каждый раз не обязательно, но если вы не уверены в причине ошибки или предпочли бы не тратить время впустую на поиск и устранение несуществующих проблем, это может оказаться полезным упражнением.

Анализ показателей построения

Собрав показатели, проанализируйте их, используя рис. 4.5 как общее руководство для определения усовершенствований, которые максимально сократят продолжительность построения. Тактики усовершенствования здесь расположены по приоритетам с использованием

Таблица 4.2. Показатели интеграционного построения

Показатель интеграционного построения	Описание
Время компиляции	Время, необходимое для компиляции программного обеспечения. Подлежит сравнению с прошлым временем компиляции
Количество *строк исходного кода* (Source Lines Of Code — SLOC)	Обозначает размер системы или ориентировочный объем кода, подлежащего компиляции
Количество и типы инспекций	Количество выполняемых инспекций разных типов. Выявите и удалите избыточные
Среднее время создания сборок	Время, необходимое для создания сборки, архива или упаковки программного обеспечения
Время проверки (по категориям)	Время, необходимое для выполнения проверок на каждом уровне: модуль, компонент и система (они описаны в главе 6, “Непрерывная проверка”)
Соотношение успешных и неудачных построений	Чтобы определить соотношение успешных и неудачных построений, разделите число последних на общее количество построений
Время инспекции	Время, необходимое для выполнения всех автоматизированных инспекций
Время развертывания	Время, необходимое для развертывания программного обеспечения в среде назначения после интеграционного построения
Время перепостроения базы данных	Время, необходимое для перепостроения базы данных
Системные ресурсы машины интеграционного построения и их использование	Увеличивая память, скорость диска и процессора, можно улучшить производительность интеграционных построений. Это поможет выяснить, имеет ли машина интеграционного построения сервер приложений, сервер баз данных и некоторые другие процессы, которые растрачивают память и скорость процессора
Загрузка системы контроля версий	Помогает выявить пиковую загруженность системы с контролем версий, продолжительность проверки и загрузки проекта с машины интеграционного построения и адекватность пропускной способности сети, процессора, памяти и дисков

следующих критериев: масштабируемость, производительность и сложность реализации. Многие решения могут зависеть от размера базового кода и ряда автоматизированных процессов построения, выполнение которых требует много времени (например, автоматизированные проверки). Вы можете задокументировать подход, рационализировать его и применить в следующий раз, когда понадобится уменьшить продолжительность построения.

Выбор и реализация усовершенствований

Вооружившись стратегией усовершенствований, вы можете приступать к их реализации.

Использование выделенной машины для интеграционного построения

Качества используемой выделенной машины интеграционного построения мы рассматривали в предыдущем разделе. Ее применение дает множество преимуществ по повышению производительности, включая снижение количества отказов построения и его ускорение.

Тактика усовершенствования	Приоритет	Масштабируемость	Производительность	Сложность
Использование выделенной машины для интеграционного построения	1	↑	↑	↓
Увеличение аппаратных возможностей машины (машин) интеграционного построения	2	↑	—	↓
Повышение производительности проверки	3	↓	↑	—
Рационализация интеграционного построения	4	↓	—	↓
Оптимизация инфраструктуры	5	—	—	↑
Оптимизация процесса построения	6	↓	—	↓
Раздельное построение компонентов системы	7	↓	—	—
Повышение производительности инспекции программного обеспечения	8	↓	↑	↓
Осуществление распределенного интеграционного построения	9	↑	↑	↑

Легенда (влияние на масштабируемость, производительность и сложность):

Высокое ↑ Среднее — Низкое ↓

Рис. 4.5. Снижение продолжительности интеграционного построения

Увеличение аппаратных возможностей машины (машин) интеграционного построения

Зачастую замена аппаратных средств машины на более быстрые — самый дешевый способ снижения продолжительности интеграционного построения. Вы, вероятно, слышали выражение, что “время процессора дешевле времени человека”; однако машину можно модернизировать лишь до определенного уровня. Ниже приведен список вопросов, который поможет вам определить, являются ли возможности аппаратных средств машины интеграционного построения максимальными.

Покупка машины интеграционного построения

Когда в 1990-х годах я работал над большим проектом, мы некоторое время использовали для интеграционного построения такую же машину, как и для разработки. Наши интеграционные построения занимали около двух часов (объем кода составлял больше миллиона строк), если не происходило никаких отказов. Вместо того чтобы смириться с фактом столь продолжительного построения, я в компании с другим разработчиком упросил руководство приобрести “самую быструю машину на рынке”. Мы провели исследования и выбрали машину с максимальными возможностями по скорости диска, памяти и процессора. Мы были убеждены, что аппаратные средства обойдутся дешевле

по сравнению со стоимостью человеко-часов ожидания завершения построения, и руководитель проекта с этим, к счастью, согласился. С новой машиной мы снизили время построения до 30 минут.

- Какова текущая скорость процессора? Имеется ли возможность повысить скорость машины? Допускает ли она переход на более быстрый процессор или конфигурацию *симметричной многопроцессорной обработки* (Symmetric Multiprocessing — SMP)?
- Какой процент доступной машине памяти используется?
- Всю ли доступную пропускную способность сети использует система?

В зависимости от ответов на эти вопросы имеются несколько способов повышения производительности и масштабируемости построения.

- Осуществить модернизацию процессора, диска или памяти.
- Перегрузить процессы на другие системы.
- Устранить из системы ненужные процессы.

Повышение производительности проверки

Даже в хорошо функционирующей системе CI выполнение множества автоматизированных проверок способно существенно увеличить время интеграционного построения. Оценка и повышение производительности таких проверок может значительно уменьшить продолжительность построения. Учет следующих факторов поможет вам повысить производительность проверки.

- Время автоматизированной проверки. Исследуйте хронометраж проверки, предоставленный средой ее выполнения.
- Для анализа некоторых областей кода проверки используйте инструмент измерения производительности. Кроме того, большинство сред проверки xUnit предоставляют утилиту, отображающую время выполнения каждой проверки.
- Используйте инструменты инспекции для анализа кода проверки и его сложности.
- Удостоверьтесь, что ваши проверки модулей на самом деле являются таковыми, а не проверками компонентов или системы. Это можно выяснить очень быстро, достаточно отключить сетевой кабель, завершить работу базы данных и запустить проверки. Проверки модулей пройдут без проблем, а остальные — нет.

После общей оценки среды проверки вы будете иметь лучшее представление о способах повышения ее производительности. Возможно несколько стратегий, включая следующие.

- Разделите автоматизированные проверки по категориям (модуль, компонент и система) и запускайте их в разное время (например, проверки модуля при каждой передаче, а проверки компонентов и системы при последующем построении). Более подробная информация о классификации проверок приведена в главе 6, “Непрерывная проверка”.
- Проведите рефакторинг проверок на основании результатов работы инструментов инспекции.
- Используйте *заглушки* (stub) или *ложные объекты* (mock object) для компонентов, проверка которых слишком сложна или продолжительна, чтобы осуществлять ее на уровне модуля. Например, ложный объект обычно реализуют для обмана интерфейса доступа к данным.

- Выделите продолжительные проверки интеграции в отдельные специализированные проверочные комплекты.
- Выполняйте проверки параллельно.
- Запускайте различные виды проверок в соответствии с типом построения: передающее построение, последующие построения, полное интеграционное построение или финальное построение.

Поэтапное построение

Как уже упоминалось, еще один подход снижения продолжительности построения заключается в выполнении облегченного построения, сопровождаемого "тяжеловесным" построением (Фаулер называет это *поэтапным построением* (staged build): передающее построение, сопровождаемое последующим построением). Рис. 4.6 иллюстрирует данный подход. При поэтапных построениях вы сначала запускаете "передающее", или облегченное интеграционное построение, в ходе которого интегрируются компоненты программного обеспечения и запускаются проверки модуля, позволяющие обнаружить наиболее очевидные проблемы. Если облегченное построение прошло успешно, запускается более полное интеграционное построение, включающее проверки компонентов и системы, инспекции, а также развертывание. Так реализуется описанная ранее в этой главе практика "ранний сбой построения".

Рис. 4.6. Процесс поэтапного построения

Исследование инфраструктуры

Вы можете обнаружить, что замедление интеграционного построения связано с инфраструктурой системы. Возможно, сеть недостаточно быстра или медленно выполняется соединение по виртуальной закрытой сети. Географически рассредоточенные системы при ненадежных аппаратных средствах и программном обеспечении также способны вызвать проблемы производительности. Исследуйте и усовершенствуйте все ресурсы инфраструктуры, поскольку это может существенно ускорить построение.

Оптимизация процесса построения

Большой объем базового кода может привести к тому, что интеграция компонентов программного обеспечения потребует значительного времени. Чтобы выяснить, связана ли проблема с размером кода или интеграцией этих компонентов, определите период времени, занимаемый этапом компиляции. Если данный период слишком большой, выполняйте инкрементное построение вместо полного.

При *инкрементном построении* (incremental build) компилируются только измененные файлы. Это может быть небезопасно, поскольку в зависимости от реализации вы можете не получить всех преимуществ CI. Эффективная система CI должна снижать риски, а следовательно, система интеграции должна была бы удалять все прежние файлы, а затем обновлять и компилировать код, чтобы гарантированно обнаруживать возможные проблемы. Таким образом, использовать инкрементное построение следует как последнее средство, после исследования и усовершенствования других областей, ведущих к замедлению процесса построения.

Инкрементное построение можно применить в нескольких областях. Например, при наличии системы Java с базовыми библиотеками DLL или совместно используемой библиотекой объектных модулей, которая редко изменяется, вполне приемлемо перестраивать такие библиотеки только один раз в день. Фактически, некоторые могли бы оспорить данный подход, заявив, что такие нечасто изменяемые библиотеки DLL и совместно используемые объекты считаются отдельным проектом CI и обращаться к ним стоит как к зависимым элементам проекта.

Раздельное построение компонентов системы

Иногда интеграционное построение занимает много времени из-за интеграции исходного кода с другими файлами. В таком случае вы можете разделить программное обеспечение на меньшие подсистемы (модули) и осуществлять построение каждой из них в отдельности.

Для индивидуального построения компонентов системы создайте отдельные проекты по каждому модулю, который может быть изолирован. Это можно осуществить внутри системы CI, достаточно сделать одну из подсистем главным проектом. Если в один из проектов внесены какие-либо изменения, то на основании зависимости другие проекты тоже перестраиваются. Рис. 4.7 демонстрирует типичную компоновку проекта при разделении на дискретные компоненты для ускорения построения.

Повышение производительности инспекции программного обеспечения

Аналогично низкой производительности проверки низкая производительность инспекции также может замедлить систему CI. Чтобы выяснить, не тормозит ли инспекция интеграционное построение, используйте следующий список вопросов.

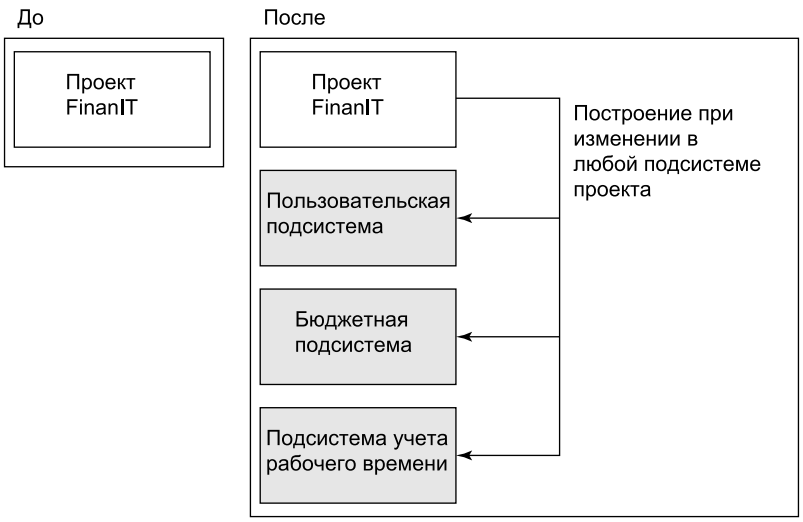


Рис. 4.7. Раздельное построение компонентов системы

- Какие показатели используются? Каждый ли показатель важен?
- Не предоставляют ли те же показатели два или больше инструмента? Это может существенно снизить производительность построения.
- Стоит ли запускать все автоматизированные инспекции при каждом построении? Возможно, некоторые инспекции можно запускать в составе вторичных или периодических построений?
- Имеются ли инспекции, которые можно запускать лишь для определенных подсистем, а не для всего базового кода?

Ниже приведен список возможных решений по увеличению производительности инспекции программного обеспечения.

- Удалите неиспользуемые и ненужные инспекции.
- Ограничьте количество повторяемых инспекций.
- Уменьшите частоту некоторых инспекций.

Осуществление распределенного интеграционного построения

Если вы имеете чрезвычайно большой объем базового кода и уже увеличили скорость процессора, памяти и диска на машине интеграционного построения, а также приняли другие меры уменьшения продолжительности построения, включая снижение частоты проверок компонентов и системы, но считаете, что построение все еще слишком продолжительно, следует обратить внимание на распределенное интеграционное построение.

Существуют инструменты интеграционного построения, которые позволяют объединить мощности нескольких машин. Такие инструменты, как BuildForge[11] и ParaBuild[12],

[11] Более подробная информация находится на сайте *www.buildforge.com*.

[12] Более подробная информация приведена по адресу *www.viewtier.com/products/parabuild/index.htm*.

предоставляют средства для распределения интеграционных построений. Данными способностями обладают и другие серверы CI, например CruiseControl, однако распределенные интеграционные построения — это сложная проблема с еще более сложным решением. Перенос части процесса построения на другую машину может подразумевать копирование больших файлов, что может даже больше замедлить построение. Перед тем как пытаться реализовать этого решение, попробуйте принять *все другие* меры по уменьшению продолжительности построения.

Переоценка

Итак, мы обсудили несколько подходов, включая повышение производительности проверки, процесса построения, модернизации аппаратных средств и проекта. Какие из улучшений оказались эффективны и какова продолжительность построений сейчас? Теперь пришло время попытаться распространить усовершенствования на остальную часть группы и решить, нужен ли дополнительный цикл улучшений. Если вы уже осуществили данный процесс один раз, то новый цикл усовершенствований пройдет быстрее и проще.

Как это будет работать у вас?

На настоящий момент вы, вероятно, вполне согласны с тем, что выполнение интеграционного построения при каждом изменении программного обеспечения позволит снизить множество рисков в проекте. Но вы можете подумать: "Это прекрасно сработало в *вашем* проекте, но это не сработает в нашем, поскольку у нас нет ни времени, ни ресурсов, ни денег" или "У нас совершенно *другой* тип проекта, вовсе не похожий на описанный здесь". Приведенные ниже вопросы и ответы основаны на ряде подобных мнений.

"Мой проект насчитывает семь миллиардов строк кода. Как это может сработать у меня?"

Да ладно, не преувеличивайте, ваш проект вряд ли имеет именно "семь миллиардов" строк кода, скажем, вы работаете над очень большим проектом и полагаете, что для внедрения CI в нем слишком много препятствий. Однако чем больше проект, тем больше количество изменений, а следовательно, тем необходимей CI. Это подобно высказыванию: "Я предпочел бы не знать о проблемах в нашем базовом коде или предпочел бы подождать, пока не забуду, над чем я работал тогда". Кроме того, внедрение CI в большой проект не займет больше времени, чем в малый. Просто в большом проекте и преимуществ будет больше, и вероятность успеха выше, и больше гибкости в работе с большим количеством элементов проекта.

Основной интерес большого проекта — это быстрое построение, а CI позволяет запускать длительные процессы периодически (или поэтапно, как описано ранее), а не непрерывно. Сюда относится проверка компонентов, системы, функций и инспекции. Разделение базового кода на отдельные составляющие также может помочь снизить продолжительность интеграционного построения.

"У меня устаревшая система, что это может мне дать?"

Если вы еще не используете систему CI, то сначала может понадобиться некоторое время на создание сценария построения, чтобы поддерживать ваш исходный код. Сценарии построения следует писать так, чтобы они могли запускаться автоматически. Но даже если вы не имеете автоматизированных проверок, вы можете начинать добавлять их для каж-

дого изменения (т.е. обнаружив дефект, пишите для него автоматизированную проверку). Затем включите запуск такой проверки в сценарий построения и выполняйте ее с составе системы CI.

"Что, если наш исходный код находится в нескольких хранилищах с контролем версий?"

Этот вопрос зачастую связан с вопросом о распределенной разработке. Давайте предположим, что вы имеете один проект в хранилище Subversion Project Management System (система управления проектом) и другой — в хранилище Financial Management System (система управления финансами). Если изменения внесены в хранилище Financial Management System, то проект в хранилище Project Management System следует перепостроить, поскольку он использует API из хранилища Financial Management System. Ваш сервер CI должен обеспечить вам возможность построения зависимости. Это запуск построения одного проекта в результате инициации построения другого.

"Наш проект распределен географически, так как мы можем заниматься CI?"

Вы имеете группы разработки, функционирующие дистанционно, и испытываете затруднения во внедрении CI? Причиной может быть медленное сетевое подключение или высокая степень защиты интеллектуальной собственности. Большинство серверов CI позволяют использовать зависимости проекта. Предположим, что в Вирджинии разрабатывается проект программного продукта, алгоритмы и некоторые компоненты которого создает группа из Калифорнии. Для хранения специальных алгоритмов компания использует хранилище с контролем версий CVS в Вирджинии. Кроме того, в Калифорнии компания установила новое хранилище Subversion. Технический руководитель в Вирджинии настраивает сервер CI CruiseControl так, чтобы поддерживать оба проекта: один в Вирджинии, второй в Калифорнии. Прежде чем интеграционное построение будет успешно выполнено для группы в Калифорнии, сервер CI запускает его в Вирджинии. Но это будет работать, только если компоненты разделены корректно.

"Мои интеграционные построения занимают слишком много времени!"

См. раздел "Выполняйте быстрое построение" ранее в этой главе.

"У нас очень часты отказы построения. Мы что-то делаем неправильно?"

Просто вы передаете в хранилище неработающий код! Возможно, он не компилируется, не проходит проверки или инспекции, либо ваши сценарии базы данных содержат ошибки. Один из способов решения данной проблемы заключается в проведении на машинах разработки закрытых построений (см. главу 2, "Введение в непрерывную интеграцию"), в максимально возможной степени имитирующих среду интеграции, перед передачей изменений в хранилище с контролем версий. Это означает, что каждый разработчик помещает последние изменения в хранилище с контролем версий, добившись успешного прохождения на машине разработки всех проверок, инспекций, а также перепостроения базы данных с проверочными данными. Это также означает, что каждый разработчик должен иметь в среде собственное "пространство", в котором выполняются те же процессы, что и при интеграционном построении. Важнейший принцип, который следует уяснить: *избежание больших построений предотвращает ошибки*. Это означает, что передаче изменений в хранилище с контролем версий должен *предшествовать* процесс интеграции и проверки на машине разработчика. Рис. 4.8 демонстрирует этапы выполнения закрытого построения перед передачей изменений в хранилище.

Рис. 4.8. Выполнение закрытого построения уменьшает вероятность ошибки интеграционного построения

"Мы не можем предоставить отдельную машину для построения"

Аппаратные средства обходятся дешевле времени сотрудников, которое может быть потрачено на поиск причин проблем интеграции. По деньгам это не дорого. Как указано в диалоге Билла и Питера ранее в этой главе, вы можете найти неиспользуемый компьютер и настроить его как машину построения. Затем, после того как группа привыкнет к преимуществам интегрированного построения, можно вложить деньги в более мощную машину. Без отдельной машины построения вы будете также тратить время на поиск проблем, вызванных тем, что разработчик забыл передать файл в хранилище с контролем версий. Это тоже требует времени, а время — деньги. Постарайтесь убедить своего "казначея" и руководство, что приобретение машины построения в конечном итоге сэкономит деньги. Здесь под "периодом окупаемости капиталовложений" следует понимать несколько недель (в зависимости от размера вашей группы), а не месяцы или годы[13]. Кроме того, вы получите качественное преимущество, воспроизводимый процесс построения, который позволит получать работоспособное программное обеспечение *в любой момент*.

"Наше программное обеспечение слишком сложно; многое приходится делать вручную" или "Мы очень заняты, нам некогда"

Это совершенно очевидная причина создания системы CI, поскольку вы, вероятно, тратите слишком много времени на выполнение избыточных процессов. Если ваше программное обеспечение сложно и имеет много зависимостей — это даже более важная причина формирования системы, которая объединяет все элементы, а также запускает комплект проверок и инспекций, гарантируя, что все будет работать правильно и *непрерывно*. Мы не имеем в виду, что вам будет легко создать воспроизводимый процесс построения, в большой инфраструктуре разработки это займет больше времени, но и его экономия окажется ощутимей.

Создать систему построения будет проще, разделив процесс на ряд малых этапов. Сначала упростите структуру каталога для хранилища с контролем версий, чтобы исход-

[13] В наших условиях, полагаю, следует учитывать различие между вашей ценой рабочего времени (т.е. зарплатой) и ценой рабочего времени у них. — *Примеч. ред.*

ный код, код проверок, файлы конфигурации и т.д. были легко доступны. Затем используйте инструмент сценариев построения, чтобы написать простой сценарий построения, который только компилирует исходный код. Теперь добавьте проверки и инспекции. Старайтесь наращивать сложность построения постепенно, не пытайтесь сделать все сразу. Фактически, именно так создавалось большинство наших систем CI. Получив награду за несколько первых этапов, у вас появится повод двигаться дальше. Продолжайте в таком же духе, работая по сценарию "немного написать, немного проверить".

"Наше программное обеспечение использует хранилище с контролем версий, но нам необходимо поддерживать несколько версий с использованием ветвления. Как нам быть?"

Это важный момент. *CI выполняется для основной линии*. Вы должны гарантировать, что она будет стабильна *всегда*. Группа разработки может быть рассредоточенной или распределенной по нескольким задачам, что существенно затрудняет коммуникацию. Использование ветвления — хорошая идея, но изменения должны применяться к основной линии.

Хотя большинство систем управления построением может запускать его для нескольких линий разработки, "интеграционное построение CI" выполняется с основной линией.

Резюме

В этой главе описаны некоторые из практик построения программного обеспечения. Построение состоит из действий, в результате которых создается работоспособное ПО: компиляция исходного кода, интеграция базы данных, проверка, инспекция, развертывание и обратная связь. Это не исчерпывающий список, есть множество других действий, которые могут стать частью процессов, запускаемых кнопкой <Integrate>.

Табл. 4.3 подводит итог практик, описанных в данной главе.

Таблица 4.3. Практики CI, обсуждаемые в этой главе

Практика	Описание
Автоматизируйте построения	Пишите сценарии построения, которые отделены от IDE. Впоследствии они будут выполняться системой CI, чтобы программное обеспечение строилось при каждом изменении в хранилище
Выполняйте построение одной командой	С учетом возможности загрузки некоторых инструментальных средств вы можете ввести одну команду и выполнить построение из вашего сценария, включая получение последнего кода и запуск всего построения
Отделяйте сценарии построения от вашего IDE	Вы должны уметь запускать автоматизированные построения без участия IDE
Централизуйте элементы программного обеспечения	Для уменьшения количества нарушенных зависимостей централизуйте все элементы программного обеспечения. Это снизит вероятность сбойных построений при перемещении на другую машину
Создайте строгую структуру каталога	Создайте однозначную, логичную структуру каталога, которая облегчит построение программного обеспечения
Ранний сбой построения	Чем быстрее обратная связь, тем раньше устранение проблемы. Выполните операции построения в таком порядке, чтобы действия с наибольшей вероятностью неудачи выполнялись *сначала*
Осуществляйте построение для каждой среды	Проводите на своей рабочей станции то же самое автоматизированное построение, что и на машине интеграционного построения, а при необходимости и для всех других сред

Окончание табл. 4.3

Практика	Описание
Используйте выделенную машину для интеграционного построения	Используйте для выполнения построения выделенную машину. Удостоверьтесь, что в области интеграции не остались прежние элементы
Используйте сервер CI	В дополнение или как альтернативу выполнению ручных интеграционных построений используйте сервер CI, такой как CruiseControl, для автоматического опроса хранилища с контролем версий на предмет изменений и запуска интеграционного построения на отдельной машине
Выполняйте интеграционное построение вручную	Запускайте последовательное интеграционное построение вручную, используя автоматизированное построение как способ уменьшения ошибок интеграционного построения. Некоторые применяют данный подход как альтернативу серверу CI
Выполняйте быстрое построение	Постарайтесь свести интеграционные построения до десяти минут, увеличив ресурсы компьютера, ограничив медленные проверки, ограничив и пересмотрев инспекции и применив поэтапные построения
Поэтапное построение	Применяйте облегченное "передающее" построение, осуществляющее компиляцию, проверку модулей и развертывание, сопровождаемое "последующим", исчерпывающим построением, которое включает проверки компонентов, системы и другие медленные проверки и инспекции

Вопросы

- Автоматизировано ли ваше построение? Действительно ли вы способны запустить построение без IDE?
- Сосредоточены ли все ваши элементы программного обеспечения в хранилище с контролем версий? Действительно ли вы способны полностью выполнить построение, получив все необходимые файлы из хранилища с контролем версий?
- Уверены ли вы, что задачи построения с наибольшей вероятностью сбоя расположены в начале ваших сценариев построения, чтобы вы быстрее получили уведомление об отказе?
- Имеете ли вы "кнопку <Integrate>" для запуска процессов построения программного обеспечения? Автоматизирована ли интеграция вашей базы данных? Проверка? Инспекция? Развертывание? Получаете ли и используете ли вы обратную связь этих процессов?
- Происходит ли ваш процесс интеграционного построения на отдельной машине?
- Какова продолжительность ваших интеграционных построений? Принимаете ли вы меры для его сокращения и улучшения обратной связи?
- Используете ли вы сервер CI для интеграции программного обеспечения? Или ваш дисциплинированный коллектив осуществляет интеграционные построения вручную?
- Как часто в вашем проекте выполняются интеграционные построения: еженедельно, ночью или ежечасно? Или непрерывно, при каждом изменении?

Часть II

Создание полнофункциональной системы CI

Глава 5

Непрерывная интеграция баз данных

Вещи не изменяются; изменяемся мы.

Генри Дэвид Торо (Henry David Thoreau)

Непрерывная интеграция базы данных (Continuous Database Integration — CDBI) — процесс ее переповторения, а также проверочных данных в любой момент, когда в хранилище с контролем версий проекта вносятся изменения.

Встречались ли вам проекты, где разработчики исходного кода и базы данных "живут в разных галактиках"? Разработчик может ждать изменений базы данных несколько дней. Ему бывает лень вносить незначительные изменения в проверочные данные или попросту страшно нарушить *совместно используемую* базу данных. Подобная ситуация не так уж редка, и применение CDBI может действительно облегчить решение некоторых из этих и многих других проблем.

Напомним тему этой книги: интеграция базы данных одной кнопкой <Integrate> (рис. 5.1). Дело в том, что основной код базы данных (DDL, DML, файлы конфигурации и т.д.), в сущности, ничем не отличается от остальной части исходного кода в системе. Фактически для интеграции базы данных ее элементы должны обладать следующим:

- возможностью расположения в системе контроля версий;
- возможностью проверки и инспекции на соответствие принятым правилам;
- возможностью создания с использованием сценария построения.

Следовательно, построение базы данных можно включить в систему CI и наслаждаться теми же преимуществами, что и для остальной части исходного кода проекта. Более того, изменения исходного кода базы данных могут запускать интеграционное построение *точно так же, как и другие изменения исходного кода.*

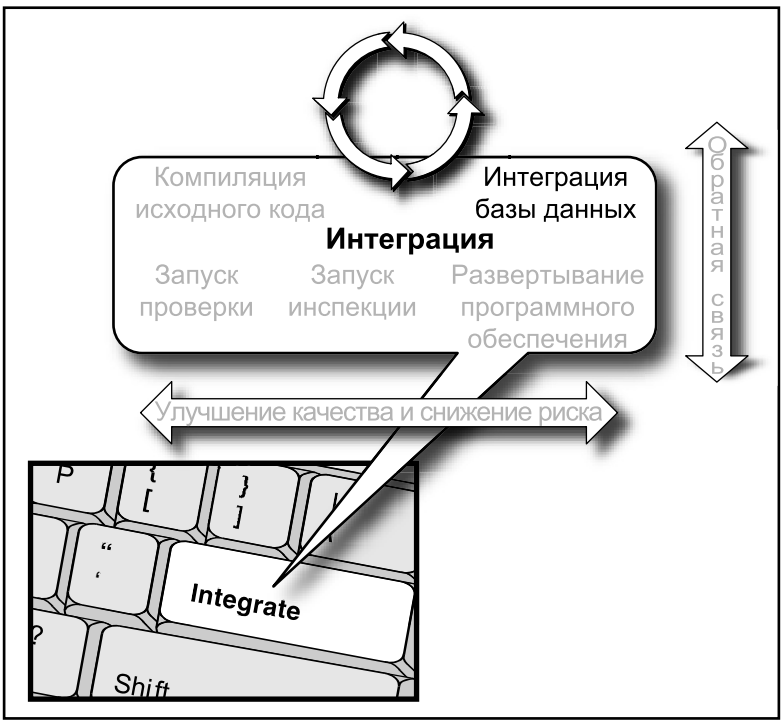


Рис. 5.1. Интеграция базы данных при помощи кнопки <Integrate>

Не все источники данных одинаковы

В некоторых проектах или частях проектов база данных может использоваться совсем не так, как в этой главе. Однако в большинстве проектов данные приходится хранить либо в текстовом файле (flat file), либо в файле XML, двоичном файле или RDBMS. Независимо от выбранного способа хранения принципы CDBI одинаковы.

В качестве первого шага автоматизации интеграции базы данных с системой CI обеспечим внедрение интеграции базы данных в процесс построения. Сценарии, используемые для построения, настройки и заполнения базы данных, должны быть доступны остальным участникам проекта, поэтому определим, какие файлы базы данных следует передавать в хранилище с контролем версий. Автоматизация процесса интеграционного построения базы данных решает только часть проблемы, поэтому следующим этапом будет организация перепостроения базы данных и проверочных данных при *каждом* изменении в программном обеспечении, а также обеспечение непрерывного процесса проверки. Если группа применяет CDBI впервые, большинству участников проекта, по-видимому, придется изменить свой стиль разработки, поэтому мы закончим главу рассмотрением эффективных практик CDBI.

Рефакторинг базы данных

Некоторые из анализируемых в этой главе тем вполне заслуживают отдельных книг[1]. В других материалах базу данных рекомендуют рассматривать лишь как другой тип исходного кода, управляемый хранилищем с контролем версий. Эта глава демонстрирует вам наиболее существенные поводы для автоматизации и запуска процессов непрерывной интеграции базы данных.

Автоматизируйте интеграцию базы данных

Во многих проектах *администратор базы данных* (Database Administrator — DBA) зачастую похож на повара столовой. Как правило, DBA имеет многолетнюю квалификацию аналитика, но зачастую тратит слишком много времени на низкоуровневые задачи команды. В результате его работа может оказаться напряженной, поэтому база данных зачастую становится узким местом разработки, а остальные участники группы вынуждены ожидать, пока DBA вносит одно незначительное изменение в базу данных за другим. Вот, несомненно, знакомый сценарий.

Нона (разработчик). Эй, Джули, вы установите для меня базу данных на совместно используемой машине разработки?

Джули (DBA). Я как раз этим и занимаюсь. Я закончу, вероятно, ближе к обеду. Вы хотели бы заполнить ее данными с прошлой недели или экспортируете сегодняшние?

Нона. Сегодняшними.

Джули. Хорошо, но я смогу сделать это только к завтрашнему утру.

Через 10 минут ...

Скотт (технический руководитель). Я не могу выполнить проверку на сервере проверки, поскольку для роли `Reviewer` нет никаких записей.

Джули. Ох... я создам проверочные записи для этой роли. Возможно, это Нона удалила их.

Скотт. Спасибо. Кстати, вы не удалили бы заодно ограничитель Y/N на столбце APPROVED в таблице PERSON? Мы хотели бы использовать в этом столбце различные флаги.

Это типичный пример из жизни DBA. И дело не только в недостаточном использовании его талантов, просто это на самом деле слабое место, особенно при непрерывном подходе, обусловленном CI. Если бы вы спросили любого DBA, чем бы он с удовольствием занимался ежедневно, то он, скорее всего, сообщил бы вам, что тратил бы время на нормализацию данных, улучшение производительности или разработку и внедрение стандартов, а не на предоставление людям доступа к базе данных или ее повторные создания с обновлением проверочных данных. В этом разделе вы увидите, как можно автоматизировать эти повторяемые задачи так, чтобы и DBA, и вся группа тратили время в основном на улучшение базы данных и повышение ее эффективности, а не на простое администрирование. Табл. 5.1 демонстрирует действия по интеграции базы данных, обычно выполняемые участником проекта. Эти действия можно автоматизировать.

Автоматизация только этих задач, связанных с базой данных, позволит вам решить лишь проблемы удаления и создания базы данных, сопровождаемого вставкой проверочных данных. В примерах настоящей главы используется Ant, но эти принципы применимы

[1] Фактически Скотт В. Эмблер (Scott Ambler) и Прамодкумар Дж. Садаладж (Pramod Sadalage) сделали это в своей книге *Рефакторинг баз данных* (Пер. с англ., ИД "Вильямс", 2007). Мартин Фаулер и Прамодкумар Садаладж писали о подобных темах в статье "Evolutionary Database Design" на сайте `www.martinfowler.com/articles/evodb.html`.

Таблица 5.1. Повторяемые действия при интеграции базы данных

Действие	Описание
Удаление базы данных	Удаление базы данных позволяет создать новую базу данных с тем же именем
Создание базы данных	Создание новой базы данных с использованием *языка определения данных* (Data Definition Language — DDL)
Вставка системных данных	Вставка всех исходных данных (например, таблиц подстановок), которые ваша система предположительно будет содержать при поставке
Вставка проверочных данных	Вставка проверочных данных в проверяемые элементы
Перемещение базы данных и данных	Переодическое перемещение схемы базы данных и данных (если вы создаете систему на основании существующей базы данных)
Установка экземпляров базы данных для нескольких сред	Установка отдельных баз данных для поддержки разных версий и сред
Модификация атрибутов столбцов и ограничителей	Модификация атрибутов столбцов таблицы и ограничителей на основании требований и рефакторинга
Модификация проверочных данных	Изменение проверочных данных в соответствии с несколькими средами
Модификация хранимых процедур (наряду с функциями и триггерами)	Модификация и проверка хранимых процедур по много раз в течение разработки (при использовании хранимых процедур это обычно необходимо для обеспечения правильного поведения программного обеспечения)
Получение доступа к различным системам	Доступ к базам данных различных систем с использованием идентификатора пользователя, пароля и идентификатора базы данных
Резервное копирование и восстановление больших наборов данных	Создание специализированных функций для особенно больших наборов данных или всех баз данных

к любой платформе построения, поддерживающей связь с базой данных. Если вы используете такую платформу построения, как NAnt, Rake или Maven, то вы вполне можете осуществить те же действия, которые демонстрирует эта глава. Листинг 5.1 содержит набор операторов SQL, создающий базу данных, ее связанные таблицы, комментарии, ограничители и хранимые процедуры. Сценарий добавляет также проверочные данные для определенной среды, например среды разработки или QA. Вы можете просто ввести команду **`ant db:prepare`**[2] в командной строке, и процесс построения выполнит задачи, указанные в табл. 5.1. Если вы хотите выполнить тот же процесс другим инструментом, например NAnt или Maven, мы предоставили дополнительные примеры на Web-сайте книги.[3]

Листинг 5.1. Файл `build-database.xml`: автоматизация интеграции базы данных при помощи Ant

```
> ant -f build-database.xml db:prepare
Buildfile: build-database.xml
```

[2] Для применения других сред из командной строки включите их в ваш сценарий построения, переопределив конфигурацию, заданную по умолчанию.

Например, в Ant это было бы *`ant -Denvironment=devqa <targetname>`*.

[3] См. *`www.integratebutton.com/`*.

```
db:create:
      [sql] Executing file: data-definition.sql
      [sql] 8 of 8 SQL statements executed successfully

db:insert:
      [sql] Executing file: data-manipulation.sql
      [sql] 60 of 60 SQL statements executed successfully

BUILD SUCCESSFUL
Total time: 20 seconds
```

Как следует из листинга, используя одну инструкцию из командной строки, можно запустить сценарии SQL, которые определяют (`db:create`) и манипулируют базой данных (`db:insert`). Более подробная информация о каждой из этих задач приведена в последующих разделах.

Рис. 5.2 демонстрирует этапы автоматизации интеграции базы данных. Все компоненты рис. 5.2 обсуждаются в следующих разделах.

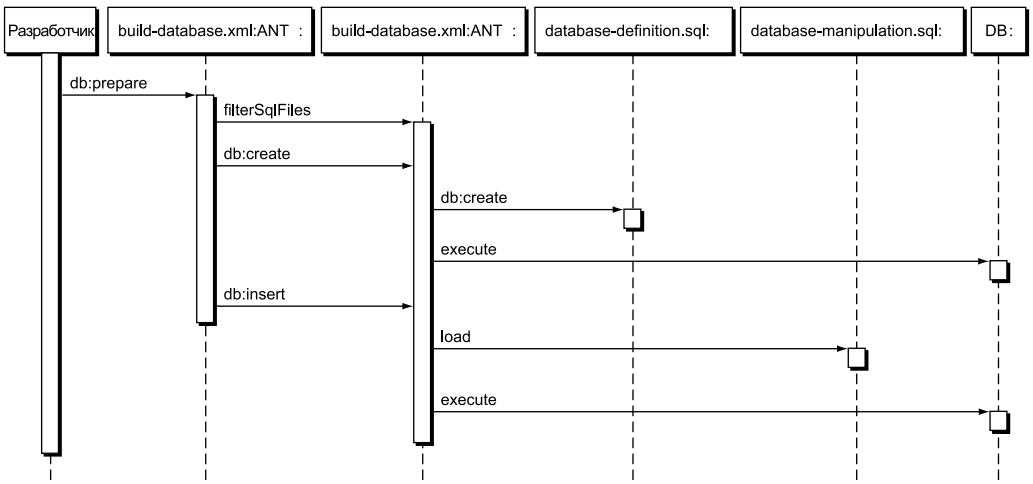


Рис. 5.2. Последовательность автоматизированной интеграции базы данных

Создание базы данных

Чтобы автоматизировать интеграцию базы данных, вы должны сначала создать ее. В этом случае вы обычно удаляете базу данных и вновь создаете ее, соблюдая целостность данных при помощи ограничителей и триггеров, а также определяя поведение базы данных при помощи хранимых процедур или функций. В листинге 5.2 для автоматизации *выполнения* этого процесса мы используем среду Ant; но, как уже упоминалось, вы можете также применять make, shell, batch, Rake, Ruby или любой другой инструмент. Обратите внимание, Ant предоставляет специальную задачу (`sql`) для выполнения сценариев SQL. Использование платформы построения, подобной Ant, позволяет выполнять действия по интеграции базы данных в результате последовательного подхода и применения зависимостей от других целевых объектов (набор задач) в сценарии. Пример в листинге 5.2 демонстрирует использование таких атрибутов задачи Ant `sql`, как `driver`, `userid` и `password`, для подключения к базе данных.

Листинг 5.2. Файл `build-database.xml`: определение базы данных при помощи сценария Ant

```
<target name="db:create" depends="filterSqlFiles" description="Create
        the database definition">
    <sql
        driver="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/"
        userid="root"
        password="root"
        classpathref="db.lib.path"
        src="${filtered.sql.dir}/database-definition.sql"
        delimiter="//"/>
</target>
```

Создание сценария многократного использования

Когда вы пишете сценарий, который планируете многократно использовать, вы можете определить атрибуты в отдельном файле, используемом в ручном и автоматизированном сценарии, чтобы не вводить их значения каждый раз.

В листинге 5.3 приведен сценарий SQL `data-definition.sql`, вызываемый сценарием Ant в листинге 5.2. В этом примере мы использовали базу данных MySQL, поэтому некоторые из команд специфичны для MySQL. Файл `data-definition.sql` несет ответственность за создание базы данных и ее таблиц, обеспечение целостности данных и применение хранимых процедур. Ниже приведен типичный порядок такого процесса создания.

1. База данных и права.
2. Таблицы.
3. Последовательности.
4. Представления.
5. Хранимые процедуры и функции.
6. Триггеры.

Порядок операторов DDL в вашем случае может быть другим, в зависимости от объекта базы данных. Например, вы можете иметь функцию, которая зависит от представления, или наоборот, так что вам может понадобиться пересмотреть последовательность.

Листинг 5.3. Файл `data-definition.sql`: пример сценария определения базы данных для MySQL

```
DROP DATABASE IF EXISTS brewery//
...
CREATE DATABASE IF NOT EXISTS brewery//

GRANT ALL PRIVILEGES ON *.* TO 'brewery'@'localhost' IDENTIFIED BY
'brewery' WITH GRANT OPTION//
GRANT ALL PRIVILEGES ON *.* TO 'brewery'@'%' IDENTIFIED BY 'brewery'
WITH GRANT OPTION//

USE brewery//
...
```

```
CREATE TABLE beer(id BIGINT(20) PRIMARY KEY, beer_name VARCHAR(50),
brewer VARCHAR(50), date_received DATE);
CREATE TABLE state(state CHAR(2), description VARCHAR(50));//
...
CREATE PROCEDURE beerCount(OUT count INT)
BEGIN
    SELECT count(0) INTO count FROM beer;
END
//
```

С технической точки зрения?

Вы можете посчитать, что проще – организовать целевые объекты и сценарии по типам определения базы данных (например, таблицы, представления и функции) или по подсистемам (например, Property и Application).

Манипулирование базой данных

После создания базы данных при помощи сценария построения ее необходимо заполнить исходными данными (например, таблицей подстановок) и данными для проверки кода, взаимодействующего с нею. Именно здесь вы предоставляете проверочные данные для вашей конкретной среды или контекста проверки. Но важней всего то, что вы можете по своему усмотрению использовать различные файлы данных SQL, поддерживающие различные среды, например разработки, проверки, QA и развертывания.

В листинге 5.4 приведен пример сценария Ant, ссылающийся на файл SQL, содержимое которого вставляется в базу данных как проверочные данные.

Листинг 5.4. Файл `build-database.xml`: манипулирование базой данных с использованием сценария Ant

```
<target name="db:insert" depends="filterSqlFiles" description="Insert
    data">
    <sql
        driver="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/brewery"
        userid="brewery"
        password="brewery"
        classpathref="db.lib.path"
        src="${filtered.sql.dir}/database-manipulation.sql"
        delimiter=";"/>
</target>
```

Сценарий SQL в листинге 5.5 представляет проверочные данные. На него ссылается код листинга 5.4. В типичном сценарии вы будете иметь намного больше записей, чем три продемонстрированные в листинге 5.5. Просто мы намерены дать вам представление о том, как зачастую выполняются сценарии SQL. Инструменты, подобные DbUnit и NDbUnit[4], также могут пригодиться при удалении и добавлении начальных данных в базу данных.

[4] Инструмент DbUnit доступен по адресу *www.dbunit.org/*, а NDbUnit — по адресу *www.ndbunit.org/*.

Листинг 5.5. Файл `data-manipulation.sql`: пример сценария манипулирования базой данных для MySQL

```
INSERT INTO beer(id, beer_name, brewer, date_received) VALUES (1,
'Liberty Ale','Anchor Brewing Company','2006-12-09');
INSERT INTO beer(id, beer_name, brewer, date_received) VALUES (2,
'Guinness Stout','St. James Gate Brewery','2006-10-23');
INSERT INTO state (state, description) VALUES('VT','Vermont');
INSERT INTO state (state, description) VALUES('VA','Virginia');
INSERT INTO state (state, description) VALUES('VI','Virgin Islands');
```

Чтобы воспользоваться преимуществами автоматизированной интеграции базы данных, вам понадобятся сценарии для вставки, модификации и удаления данных. Эти сценарии манипулирования данными выполняются как часть общего процесса построения. Далее мы обсудим, как их связать в единый *сценарий взаимодействия* (orchestration script).

Создание сценария взаимодействия для базы данных

Интеграционный сценарий взаимодействия для базы данных выполняют операторы DDL и DML (Data Manipulation Language — *язык обработки данных*). Листинг 5.6 демонстрирует сценарий Ant, который использует задачу `sql` для вызова файлов `data-definition.sql` и `data-manipulation.sql`, приведенных в листингах 5.3 и 5.5. Этот сценарий взаимодействия входит в состав высокоуровневого процесса построения и интеграции.

Листинг 5.6. Файл `build-database.xml`: интеграционный сценарий взаимодействия базы данных с использованием Ant

```
<target name="db:prepare" depends="db:create, db:insert"/>
<target name="db:create">
...
<target name="db:insert" depends="filterSqlFiles">
...
```

Вы на автопилоте?

В ходе автоматизации интеграции базы данных вы можете столкнуться с несколькими сложностями. Выполняя действия по интеграции базы данных вручную, очень просто неумышленно накопить ошибки. Старайтесь не делать этого. Как выразились Эндрю Хант (Andrew Hunt) и Дэвид Томас (David Thomas) в своей книге *The Pragmatic Programmer*: “Не повторяйтесь, поддерживайте свои сценарии построения “сухими”[5]. Простейшая форма “дублирования” возможна, когда мы вырабатываем привычку щелкать в GUI мастера, предоставляемого производителем базы данных, забыв об интерфейсе командной строки, где можно сделать то же самое, запустив сценарий. Другая потенциальная проблема — это тенденция накапливать много изменений DDL/DML перед передачей в хранилище с контролем версий. Изменения базы данных могут быть весьма глубокими, поэтому старайтесь делать маленькие, инкрементные изменения базы данных; так их будет проще проверять и отлаживать.

[5] Игра слов Don’t Repeat Yourself (не повторяйтесь) — DRY и “DRY” — сухой. — *Примеч. ред.*

Используйте локальное пространство базы данных

Существенной проблемой многих рабочих проектов программного обеспечения является изменение структуры базы данных. В большинстве проектов, которые я видел, обычно совместно используется одна база данных, в которую все разработчики вносят изменения. При этом изменения, вносимые одним разработчиком, могут неблагоприятно сказаться на работе других участников группы, приводя к отказу даже при закрытом построении у каждого разработчика (если их проверки являются частью построения). Подобно тому как разработчики имеют собственное локальное "пространство", позволяющее изолировать их код от кода других разработчиков, было бы неплохо также иметь локальное "пространство базы данных", не так ли?

Несколько экземпляров базы данных

Вы не можете предоставить каждому разработчику отдельную базу данных? Тогда вы могли бы обеспечить каждого из них отдельной схемой на центральном сервере баз данных или использовать один из бесплатных, облегченных эквивалентов базы данных с открытым исходным кодом. Кстати, у большинства популярных RDBMS есть бесплатные версии для разработчиков.

Еще одно немаловажное преимущество, которое вы получаете, автоматизировав интеграцию базы данных, заключается в том, что каждый участник группы сможет создать локальный экземпляр базы данных на своей рабочей станции, после чего сформировать "пространство" базы данных, чтобы вносить и проверять ее изменения независимо от других сотрудников. Если интеграция базы данных задана сценарием, то создание нового экземпляра последней — дело одной кнопки; и наоборот, если вы не автоматизируете интеграцию базы данных, то повторно создать ее и запустить проверки на рабочей станции будет труднее. Рис. 5.3 демонстрирует использование локального экземпляра базы данных каждым разработчиком.

Применяя автоматизированную интеграцию базы данных, вы можете получить последнюю версию своих сценариев базы данных наряду с исходным кодом приложения. Каждый разработчик способен создать локальный экземпляр базы данных, модифицировать ее версию на своей рабочей станции, проверить изменения и передать их обратно в хранилище. Эти изменения будут интегрированы с остальной частью программного обеспечения и проверены системой CI. Когда другой разработчик станет обновлять свое закрытое рабочее пространство изменениями из хранилища, изменения базы данных будут скопированы на его рабочую станцию вместе с другими изменениями исходного кода. В результате его следующее закрытое построение пройдет с учетом изменений в локальном экземпляре базы данных.

Поддержка нескольких сред базы данных

Следующий логический шаг после применения локального пространства базы данных — это создание ее различных экземпляров для поддержки нескольких сред. Например, вы могли бы создать базу данных, содержащую все ваши переносимые рабочие данные. Если в базе данных, например, слишком много записей, то вы вряд ли захотите переносить их в рабочую локальную базу данных. Обычно это относится только к коду DML (изменение данных), а *не* к коду DDL (операторы создания, изменения и удаления базы данных). Автоматизируя интеграцию базы данных, вы можете изменять параметры сце-

нария построения так, чтобы включить данные, поддерживающие эти среды. Таким образом вы сможете, выполнив одну команду, предоставить данные для различных сред базы данных. То же относится и к версиям. Вы можете захотеть проверять новый код на предыдущей версии базы данных. Использование автоматизированной интеграции базы данных обеспечит эффект “нажатия кнопки <Integrate>”.

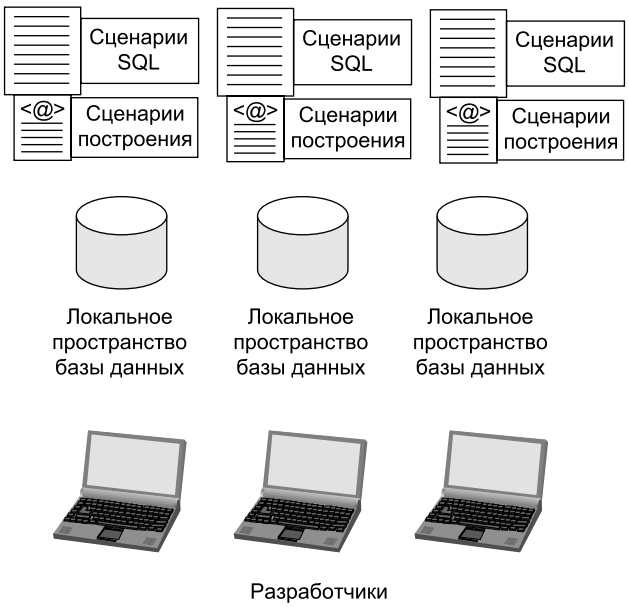


Рис. 5.3. Каждый разработчик использует локальное пространство базы данных

Следующий раздел рассматривает причины и подходы к использованию хранилища с контролем версий для интеграции базы данных.

Применяйте хранилище с контролем версий для совместного использования элементов базы данных

Совместное использование сценариев интеграции базы данных — весьма полезная, а главное, простая и понятная практика. Все элементы программного обеспечения должны находиться в хранилище с контролем версий, куда относятся и все компоненты базы данных. Такими элементами могли бы быть следующие:

- библиотеки DDL для удаления и создания таблиц, представлений, ограничителей и триггеров;
- хранимые процедуры и функции;
- схемы связи элементов;
- проверочные данные для различных сред;
- специфические конфигурации базы данных.

Довольно часто в проекте приходится создавать "с нуля" всю базу данных, причем с использованием сценария, находящегося в хранилище с контролем версий (для больших наборов данных вы можете использовать сценарии их экспорта, а не построчные сценарии DML), поэтому необходимо обеспечить такую возможность. Если вы расположили все элементы базы данных в хранилище с контролем версий, то получаете историю всех изменений базы данных и сможете запускать с последним кодом ее предыдущие версии (и предыдущие версии кода тоже). Это снижает также задержки в проектах, связанные с тем, что все разработчики вынуждены обращаться к DBA по любому поводу. Хранение элементов базы данных в одном месте позволяет вносить изменения в столбцы базы данных, выполнять закрытое построение на своей машине, а затем передавать изменения системе с контролем версий и узнавать о результате выполнения интеграционного построения по обратной связи.

Иногда в процессе разработки база данных подвергается крупномасштабным изменениям. В большинстве случаев эти изменения требуют усилий нескольких участников группы и много времени. В таких случаях имеет смысл создать *ветвление задачи* (task branch)[6], чтобы передать изменения обратно в хранилище с контролем версий, а не прерывать основную линию и замедлять работу остальной части группы. Без CDBI такие частые крупномасштабные модификации базы данных будет выполнять DBA, а он может оказаться наименее подходящим сотрудником для одновременного внесения в базу данных всех изменений, включая зависимый исходный код приложения, связанный с этим код проверки и совместно используемые сценарии, поскольку он может плохо знать исходный код, который пишут разработчики.

Точно так же как и исходный код, база данных нуждается в четкой структуре каталога. Определите расположение элементов базы данных (вероятно, где-нибудь в каталоге `implementation` или `construction`, где размещен исходный код). В каталоге баз данных создайте подкаталоги для каждого из типов ее объектов и сред. Листинг 5.7 демонстрирует структуру каталога `implementation` (подразумевается использование базы данных MySQL).

Листинг 5.7. Пример каталога `implementation`

```
implementation
    bin
    build
        filtered-sql
    config
        properties
        xml
    database
        migration
    lib
        mysql
    src
    tests
tools
    mysql
```

[6] В книге *Software Configuration Management Patterns* Стивен П. Беркзук (Stephen P. Berczuk) и Бред Апплетон (Brad Appleton) описывают ветвление задач так: "Часть вашей группы выполняет склонную к сбоям задачу, не вынуждая остальную часть группы плясать вокруг них ..."

Подобно исходному коду выберите для базы данных подходящую структуру каталога, которая однозначно идентифицирует объекты, облегчив их создание и изменение.

Структура каталога и поддержка сценариев

На первый взгляд может показаться, что структура каталога не столь важна, однако остерегайтесь ее частых изменений, поскольку вам придется потратить дополнительное время на модификацию сценариев, чтобы они соответствовали внесенным изменениям.

Теперь, когда вы автоматизировали свои действия по интеграции базы данных и ее проверке в хранилище с контролем версий, давайте сделаем процесс непрерывным, запускаемым при каждом изменении программного обеспечения.

Непрерывная интеграция базы данных

Вот где "масло на дороге разлито". Причина автоматизации, совместного использования и интеграции процесса построения базы данных заключается в возможности сделать все эти процессы непрерывными. При использовании CDBI база данных и исходный код синхронизируются по много раз в день. Когда вы передаете изменения базы данных в хранилище с контролем версий, система CI обрабатывает их примерно так: она получает полную копию исходного кода системы, включая определение данных базы данных и сценарии манипулирования; вновь создает базу данных из исходных элементов; интегрирует другой ваш исходный код; а затем выполняет автоматизированные проверки и инспекции, чтобы удостовериться в отсутствии дефектов базового кода и системы. На рис. 5.4 показано, как внесенные каждым разработчиком изменения синхронизируются при интеграционном построении на базе основной линии в хранилище с контролем версий.

Рис. 5.4 демонстрирует, что изменения, внесенные Майком в 10 часов, *и* изменения, внесенные Сэнди в 10:15, включаются в интеграционное построение, проходящее в 10:30. Машина интеграционного построения использует для синхронизации и проверки изменений в ходе операции единый исходный материал, предоставляемый хранилищем с контролем версий.

Если вы уже автоматизировали интеграцию базы данных и включили это в сценарии построения, то организовать *непрерывный* запуск не составит труда. Ваши задачи интеграции базы данных, наряду с остальной частью действий построения, должны запускаться одной командой (например, `target` у Ant или NAnt). Для их непрерывного запуска достаточно удостовериться, что команды этих задач выполняются как часть автоматизированного построения.

Обеспечьте разработчикам возможность модифицировать базу данных

Каждый разработчик должен иметь возможность модифицировать любой из сценариев базы данных. Это не означает, что он *будет* модифицировать эти сценарии, поскольку не всякий коллега обладает необходимым для этого опытом. В связи с тем, что каждый разработчик имеет свое собственное пространство базы данных, любой из них может модифицировать локальную базу данных, а затем передавать изменения в хранилище с контролем версий. Это облегчает работу DBA и позволяет разработчикам вносить необходимые изменения. DBA может оценивать новые изменения в хранилище, просматривая результаты интеграционных построений или сотрудничая с разработчиками, если построение прервалось.

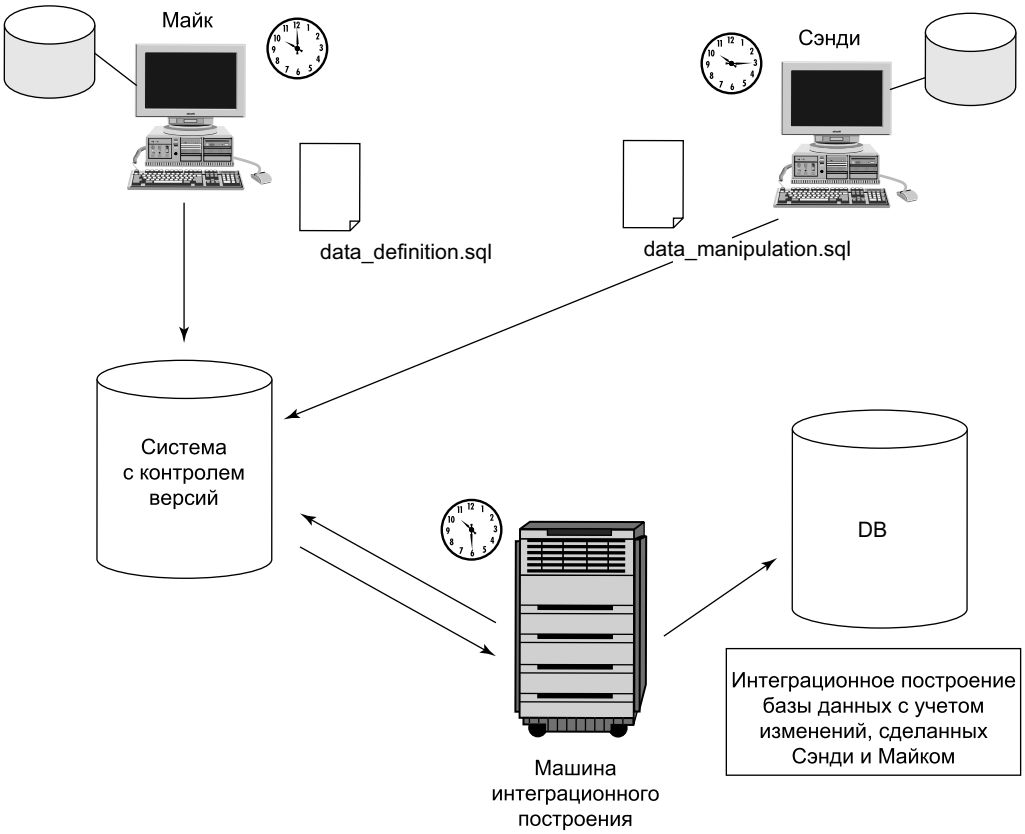


Рис. 5.4. Единый источник для изменений базы данных

Как говорится, у кого больше прав, с того и больше спрос. Изменение основной структуры базы данных может иметь далеко идущие последствия для системы. Разработчик, делающий это, должен нести ответственность за полную проверку перед передачей изменений. Мы полагаем, что у современных разработчиков достаточно знаний о базах данных и их сценариях, а DBA может "присматривать" за изменениями, вносимыми в систему.

Исправляйте сбойные построения всей группой

При работе с базой данных, так же как и с другим исходным кодом, вполне могут произойти сбойные построения из-за ошибки базы данных. Безусловно, ошибки могут наблюдаться в любой части построения: в исходном коде, при развертывании, проверке, инспекции, а также в базе данных. При использовании CDBI интеграция базы данных — это лишь еще одна часть построения, поэтому при любом его нарушении приоритетной задачей группы является устранение сбоя. Выигрыш от этого не малый, устранив ошибку и интегрировав элементы, можно гарантировать, что данная конкретная проблема больше не повторится.

Сделайте DBA участником группы разработчиков

Устраните барьеры между участниками группы базы данных и остальной частью группы разработки. Возможно, вы уже сделали это, но "граница" между DBA и разработчиками программного обеспечения зачастую прослеживается. Как уже отмечалось, рассматривайте код базы данных как любой другой исходный код. То же самое относится ко всем участникам группы. Это, вероятно, наиболее спорная из практик CDBI. Мы работали с группами, которые использовали CDBI с DBA в составе группы разработки, и видели также более традиционный подход, когда DBA принадлежал к другой группе, т.е. к группе базы данных. CDBI срабатывала в обоих случаях, но значительно лучше, когда DBA являлся частью группы.

Некоторые могут задаться вопросом: "Если DBA больше не удаляет и не создает заново таблицы и системы проверки и не предоставляет доступ, то что он делает"? Простой ответ: "Теперь он может делать свою работу!" — т.е. уделять больше времени высокоуровневым задачам, таким как повышение производительности базы данных и SQL, нормализации данных и другим дополнительным усовершенствованиям.

Интеграция базы данных и кнопка <Integrate>

В оставшейся части книги рассматриваются другие составляющие элементы кнопки <Integrate>: непрерывная проверка, инспекция, развертывание и обратная связь. В этом разделе описаны некоторые из проблем, связанные с данными практиками при интеграции базы данных.

Проверка

Подобно исходному коду вы должны проверять и базу данных. Подробно проверка будет рассматриваться в главе 6, "Непрерывная проверка". Существуют инструментальные средства, которые можно использовать для проверки базы данных, например PL/Unit, OUnit для Oracle и SQLUnit. Ваша база данных может обладать поведением, определяемым хранимыми процедурами или функциями, которые также следует проверить в составе сценария построения наравне с поведением другого исходного кода. Вы можете также захотеть проконтролировать взаимодействие ограничителей, триггеров и границ транзакций в ходе проверок защиты прикладных данных.

Инспекция

Подобно другому исходному коду необходимо инспектировать источник данных. Сюда относится не только код DDL, но и ссылки, а также проверочные данные. Имеются инструменты, которые вы можете задействовать в автоматизированном процессе построения, чтобы не выполнять эти инспекции вручную. При инспекции базы данных имеет смысл предпринять следующее.

- Проверьте достаточность производительности данных, запустив `set explain` для правил вашего проекта, чтобы оптимизировать запросы SQL.
- Проанализируйте данные, чтобы гарантировать их целостность.

- Для выявления наиболее популярных запросов используйте инструмент регистрации SQL. Эти запросы могли бы стать кандидатами на хранимые процедуры.
- Проверьте соответствие стандартам и соглашениям об именовании данных.

Развертывание

Как уже упоминалось, задача CDBI заключается в обработке исходного кода базы данных и другого исходного кода одинаковым способом. Непрерывный процесс развертывания устанавливает и проверяет экземпляры базы данных, точно так же как и другой код в различных средах (например, на сервере приложений). При необходимости перейти с одной базы данных в другую вы сможете лучше проконтролировать процесс перехода, включив его в непрерывный процесс или процесс по расписанию.

Обратная связь и документация

Внедрение непрерывной обратной связи и CDBI в систему CI позволяет узнавать о неудаче построения из-за последних изменений базы данных. По умолчанию большинство систем CI посылает информацию о состоянии построения всем вносившим изменения в хранилище с контролем версий. Точно так же как и в случае с исходным кодом, система CI сообщает о проблеме с базой данных тем, кто вносил в нее изменения, чтобы они могли оперативно принять меры по их устранению.

Документация относится к коммуникации, т.е. это та информация о базе данных, которую вы хотите сообщать другим участникам проекта или вашему клиенту. *Диаграмма сущностей и связей* (Entity Relationship Diagram — ERD), а также словарь данных — превосходные кандидаты на создание в ходе непрерывного процесса построения, возможно, при последующем построении (как описано в главе 4, “Построение программного обеспечения при каждом изменении”).

Резюме

Эта глава продемонстрировала, что элементы базы данных — это такой же исходный код, как и остальной. Следовательно, к нему применимы те же принципы.

- Автоматизируйте интеграцию базы данных, используя организованные сценарии построения, которые выполняются либо непрерывно, либо после любого изменения базы данных или ее исходного кода.
- Обеспечьте единый источник для элементов базы данных, помещая их в хранилище с контролем версий.
- Проверьте и проинспектируйте код и сценарии базы данных.
- Измените способы разработки базы данных так, чтобы вся ее интеграция осуществлялась сценарием построения, а все ее элементы проверялись в хранилище, и чтобы все разработчики (работающие с базой данных) имели свое пространство базы данных.

Табл. 5.2 подводит итог практик, описанных в данной главе.

Таблица 5.2. Практики CI, обсуждаемые в этой главе

Практика	Описание
Автоматизируйте интеграцию базы данных	Осуществляйте перепостроение базы данных и вставку проверочных данных в ходе автоматизированного построения
Используйте локальное пространство базы данных	Все разработчики должны иметь свою собственную копию базы данных, которая может быть создана при помощи сценария SQL. Она может располагаться на их рабочих станциях или даже на совместно используемом сервере разработки
Применяйте хранилище с контролем версий для совместного использования элементов базы данных	Передавайте ваши сценарии DDL и DML в хранилище с контролем версий, чтобы другие разработчики могли выполнять те же сценарии для перепостроения базы данных и проверочных данных
Обеспечьте разработчикам возможность модифицировать базу данных	Устраните узкие места DBA, возникающие из-за того, что изменения в базу данных вносит только один или два человека. Предоставьте разработчикам возможность изменять сценарии DDL и DML, а затем передавать их в хранилище с контролем версий
Сделайте DBA участником группы разработчиков	Для гарантии целостности убедитесь, что DBA может запускать тот же сценарий автоматизированного построения, включающий перепостроение базы данных, который используют другие разработчики. Сделайте DBA участником группы разработки, чтобы совместно использовать опыт и извлечь пользу и для разработки базы данных, и для разработки кода

Давайте посмотрим, что делают Джули, Скотт и Нона теперь, когда они используют CDBI.

Нона (разработчик). Мне нужно обновить проверочные данные. Что я должна сделать?

Скотт (технический руководитель). Просто выполните в командной строке команду **`ant db:refresh`**. Но прежде чем сделать это, получите последние изменения из хранилища Subversion, введя команду **`ant scm:update`**, поскольку я внес несколько изменений в таблице `USER` базы данных и в исходном коде, который использует это изменение.

Джули (DBA). Ребята, вам помощь нужна?

Скотт. Да, у нас проблема с производительностью одного из запросов. У вас есть время посмотреть его? Кроме того, я думаю, необходимо денормализовать таблицу `PRODUCT`. Не могли бы вы смоделировать ее изменения, прототипы изменения кода DDL, а также установить ветвление кода, чтобы Нона могла модифицировать свой код для ваших изменений? Если модификации вам понравятся, объедините ветвь и передайте все в хранилище Subversion, чтобы они выполнялись в составе интеграционного построения. Спасибо, Джули.

Нона. ... Понятно, Скотт. Мы должны использовать проверочную базу данных вместо рабочей?

Скотт. Да, просто выполни команду `ant –Denvironment=test db:refresh`.

Разработчики и DBA зачастую играют роли, которые кажутся несовместимыми или существенно разными, но теперь они непрерывно работают над той же задачей и совместно выполняют большее количество задач, требующих анализа или проектирования.

Вопросы

Эти вопросы помогут вам определить уровень автоматизации и непрерывной интеграции базы данных.

- Действительно ли вы способны повторно создать базу данных в ходе автоматизированного процесса построения? Можете ли вы перепостроить базу данных "нажатием одной кнопки?"
- Передаются ли сценарии автоматизации интеграции базы данных (построения и SQL) в хранилище с контролем версий?
- Действительно ли *каждый* участник вашего проекта способен повторно создать базу данных, используя автоматизированный процесс построения?
- Действительно ли вы способны в течение разработки вернуться к предыдущим версиям базы данных, используя хранилище с контролем версий?
- Действительно ли ваш процесс интеграции базы данных непрерывен? Действительно ли изменения кода программного обеспечения интегрируются и проверяются совместно с последней базой данных при каждом внесении изменений в хранилище с контролем версий?
- Выполняете ли вы проверки хранимых процедур и триггеров базы данных?
- Допускает ли ваш автоматизированный процесс интеграции базы данных настройку? Действительно ли вы способны изменить идентификатор пользователя, пароль, уникальный идентификатор базы данных, размер пространства таблиц и т.д., используя единый файл конфигурации?

Глава 6

Непрерывная проверка

Практика ведет к совершенству.

Английская пословица

Надежный (прилагательное): получение одинаковых результатов при нескольких испытаниях.[1]

Принцип технических систем свидетельствует, что надежность линейной системы определяет долговечность *каждого* из компонентов. Представим, например, систему с тремя компонентами, подобную продемонстрированной на рис. 6.1.

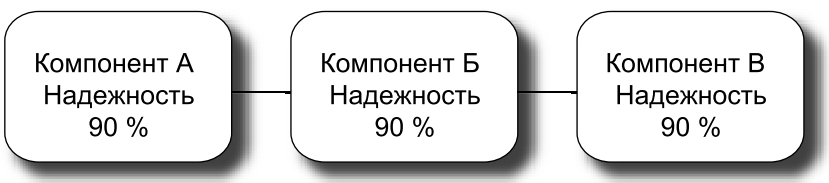


Рис. 6.1. Система из трех компонентов

[1] См `www.m-w.com/cgi-bin/dictionary?va=reliable`.

Надежность каждого компонента этой типовой системы измерена и составляет у всех 90% (не важно, как было определено это значение). Если вы не системный инженер, то, вероятно, так и будете полагать. Однако ответ не верен, на самом деле: 0,90 * 0,90 * 0,90 *равно* 0,73. Таким образом, общая надежность данной системы составит лишь 73%.

Вы бы проехали по мосту с надежностью 73%? Вы пользовались бы ручкой, которая пишет в 73% случаев или выбросили бы ее? Мы подразумеваем, что большинство мостов, по которым мы ездим, имеют стопроцентную надежность, а большинство используемых нами ручек будут писать всегда, пока не кончатся чернила. Чтобы получить такую надежность, мостостроители и изготовители ручек гарантируют качество *мельчайших деталей*, поскольку это единственный способ обеспечить надежность в целом.

Вот почему в 1970-х годах сбыт японских автомобилей начал затмевать сбыт автомобилей, выпускаемых в США. Японские изготовители нашли и применили этот принцип, подняв надежность своих марок на более высокий уровень, нежели у американских конкурентов. Японские производители поняли, что они должны гарантировать качество вплоть до *самых мельчайших и на первый взгляд кажущихся незначительными компонентов*.

Теперь представьте программную систему (она, между прочим, нелинейна, а это, по существу, означает, что вы должны также учесть надежность интерфейса или связи между всеми объектами). Вряд ли кто-то из вас работал над программной системой из трех компонентов (т.е. объектов), как на рис. 6.1. Большинство таких систем имеет сотни, если не тысячи, объектов! Линейная система из 100 компонентов с надежностью 99% обладает надежностью всего 37%.

Если вы хотите создать приложение, *соглашение об уровне услуг* (Service Level Agreement — SLA) которого подразумевает 100% (или около того) надежности, вам придется гарантировать абсолютное качество каждого объекта. Если вы не можете измерять и обеспечивать надежность на самом низком уровне, вы не сможете это сделать и на уровне системы. Но именно это в значительной степени и определяет профессионализм в создании и предоставлении программного обеспечения. Разработайте и создайте приложение, а затем передайте его группе *гарантии качества* (Quality Assurance — QA), которая проведет проверку на индивидуальном и системном уровнях и неизбежно найдет некоторое количество дефектов. На некоем этапе вы передадите систему своим клиентам, которые, как не удивительно, тоже найдут дефекты, иногда в ущерб прибыли, вашей репутации или тому и другому вместе взятым.

Таким образом, если вам необходимо построить действительно качественную программную систему, следует гарантировать надежность на уровне объектов, что достигается только при успешной проверке модуля. В противном случае можно даже не надеяться построить высоконадежное приложение. Безусловно, одни проверки модулей не гарантируют надежности объектов. Проверка должна *эффективно* охватывать все случаи использования объекта; кроме того, проверки следует выполнять *часто*.

Поскольку объекты в программной системе взаимодействуют друг с другом, проверки следует запускать *в любой момент* и *каждый раз* при изменении системы. Организация непрерывной проверки в вашей системе CI предоставляет такую возможность. Используя иллюстрацию кнопки <Integrate>, рис. 6.2 демонстрирует, на каком этапе процесса реализации полностью автоматизированного построения и системы CI мы находимся.

Автоматизируйте проверки модуля

Под термином “проверка модуля” (“unit test”) зачастую подразумевают множество вещей. Это вносит беспорядок, особенно когда люди начинают утверждать, что их проверки

модуля "слишком продолжительные, чтобы их запускать". Выработав общий словарь для различных проверок, можно эффективней категоризировать их по группам, различая которые будет проще создать систему CI, способную ускорить процесс построения.

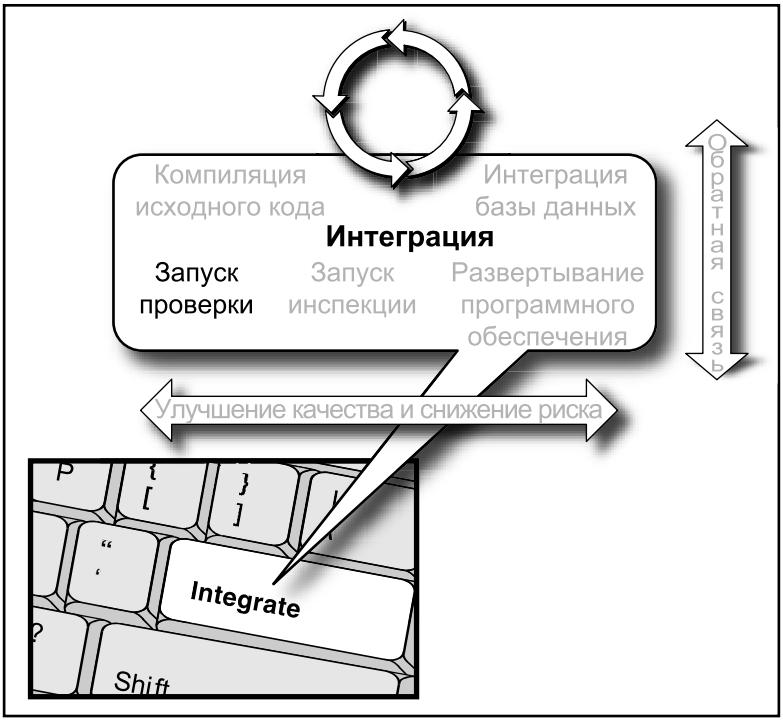


Рис. 6.2. Кнопка <Integrate> — запуск автоматизированных проверок разработчика

Проверка модуля — это проверка поведения малых элементов программной системы, которыми зачастую являются отдельные классы. Тем не менее иногда проверка модуля распространяется на несколько классов, поскольку они жестко связаны между собой.

Листинг 6.1 демонстрирует проверку модуля, написанную для среды TestNG, которая ориентирована на аннотации. Обратите внимание на комментарий `@testng.test` стиля Javadoc в методе `starPatternTest`. Данный случай проверки с использованием возможности `assert` Java 1.4 позволяет удостовериться, что класс `RegexPackageFilter` правильно фильтрует строки по схеме регулярного выражения.

Листинг 6.1. Изолированная проверка модуля с использованием среды TestNG

```
public class RegexPackageFilterTestNG {
    /**
     * @testng.test
     */
    public void starPatternTest() throws Exception{

        Filter filter = new RegexPackageFilter("java.lang.*");

        assert filter.applyFilter("java.lang.String"):
```

```
            "filter returned false";

        assert !filter.applyFilter("org.junit.TestCase"):
            "filter returned true for org.junit.TestCase";
    }
}
```

Некоторые проверки модуля требуют минимума внешних зависимостей, которые могут быть *только* другими классами. Сами по себе эти зависимые классы довольно просты и не имеют глубоких объектных зависимостей. Иногда проверки модуля используют даже *ложный объект* (mock), который, будучи простым объектом, заменяет реальный, более сложный объект. Если зависимый объект сам зависит от внешнего элемента (например, файловой системы или базы данных), который не заменяется ложным объектом, то проверка становится проверкой компонента (рассматривается далее).

Листинг 6.2 демонстрирует написанный в среде Ruby пример проверки модуля, проверяющей режим фильтрации ввода. Такая проверка все еще считается проверкой модуля несмотря на то, что в ней используются два класса, `RegexFilter` и `SimpleFilter`, поскольку здесь применяется только один ввод для проверки режима.

Листинг 6.2. Изолированная проверка модуля с использованием среды Ruby

```
require "test/unit"
require "filters"

class FiltersTest < Test::Unit::TestCase

    def test_regex
        fltr = RegexFilter.new(/Google|Amazon/)
        assert(fltr.apply_filter("Google"))
    end

    def test_simple
        fltr = SimpleFilter.new("oo")
        assert(fltr.apply_filter("google"))
    end

    def test_filters
        fltrs = [SimpleFilter.new("oo"), RegexFilter.new(/Go+gle/)]
        fltrs.each{ | fltr |
            assert(fltr.apply_filter("I love to Goooogle on the
↳Internet"))
        }
    end
end
```

Ключевой аспект для проверок модуля — это отсутствие внешней зависимости (такой, например, как база данных), которая имеет тенденцию увеличивать период установки и запуска проверки. Проверки модуля могут быть созданы и запущены на раннем этапе разработки (т.е. в первый день). В связи с коротким периодом между написанием кода и проверкой результатов проверка модуля — это наиболее эффективный способ отладки.

Автоматизируйте проверки компонента

При *проверке компонента* (component test) или подсистемы проверяют части системы. Это может потребовать ее полной установки или некоторых внешних зависимостей, таких как база данных, файловая система или сетевые конечные точки, чтобы к ним можно было обращаться. В результате проверок выясняется, как компоненты взаимодействуют друг с другом и насколько это соответствует тому, что ожидалось. Типичная проверка компонента требует, чтобы основная база данных была запущена и доступна. Поскольку при каждом случае проверки исследуются большие объемы кода, эти проверки проходят дольше проверок модуля.

Листинг 6.3 демонстрирует типовую проверку компонента, которая использует среду DbUnit для обращения к базе данных и последующей попытке их поиска на основании содержимого базы данных. Среда DbUnit использует файлы XML, которые она читает, а затем вставляет данные в соответствующие таблицы базы данных.

Листинг 6.3. Проверка компонента с использованием DbUnit

```
public class DefaultWordDAOImplTest extends DatabaseTestCase {
    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSet(new File("test/conf/wseed.xml"));
    }

    protected IDatabaseConnection getConnection() throws Exception {
        final Class driverClass =
              Class.forName("org.gjt.mm.mysql.Driver");
        final Connection jdbcConnection =
        DriverManager.getConnection(
              "jdbc:mysql://localhost/words",
              "words", "words");
        return new DatabaseConnection(jdbcConnection);
    }

    public void testFindVerifyDefinition() throws Exception{
        final WordDAOImpl dao = new WordDAOImpl();
        final IWord wrd = dao.findWord("pugnacious");
        for(Iterator iter =
                          wrd.getDefinitions().iterator();
                          iter.hasNext();){
            IDefinition def = (IDefinition)iter.next();
            TestCase.assertEquals(
                "def is not Combative in nature; belligerent.",
                "Combative in nature; belligerent.",
                def.getDefinition());
        }
    }

    public DefaultWordDAOImplTest(String name) {
        super(name);
    }
}
```

Проверки на уровне компонентов используют большее количество зависимостей, чем проверки модуля, но все еще не обязательно столько, сколько высокоуровневые проверки системы (рассматриваются далее). В ходе проверки на уровне компонентов код исследуется при помощи API, но они могут и *не быть* предоставлены клиентам. В листинге 6.3 объект на слое *объекта доступа к данным* (Data Access Object — DAO), по существу, проверяется через предоставляемый интерфейс. Другой пример проверки компонента — исследование действия класса в архитектуре Struts при помощи среды StrutsTestCase, как представлено в листинге 6.4. Эта проверка, безусловно, требует, чтобы база данных работала; однако Web-контейнер заменен ложным объектом и исследуемые API не обязательно предоставлять клиентам.

В листинге 6.4 среда StrutsTestCase была объединена с DbUnit, чтобы предоставить и функциональные возможности доступа к базе данных, и ложный контейнер. Класс `DeftMeinMockStrutsTestCase` — это шаблон, который требует, чтобы метод `getDBUnitDataSetFileForSetUp` был реализован.

Листинг 6.4. Проверка компонента с использованием StrutsTest

```
public class ProjectViewActionTest extends DeftMeinMockStrutsTestCase
{
    public void testProjectViewAction() throws Exception {
        this.addRequestParameter("projectId", "100");
        this.setRequestPathInfo("/viewProjectHistory");
        this.actionPerform();
        this.verifyForward("success");

        Project project = (Project)this.getRequest()
            .getAttribute("project");
            assertNotNull(project);
            assertEquals(project.getName(), "DS");
    }

    protected String getDBUnitDataSetFileForSetUp() {
        return "dbunit-seed.xml";
    }

    public ProjectViewActionTest(String name) {
        super(name);
    }
}
```

Данный тип проверки называют также *проверкой интеграции* (integration test). Различие между ним и проверкой системы заключается в том, что проверка интеграции (или проверка компонента, или проверка подсистемы) не всегда исследует *открытые* API. Например, проверка системы исследовала бы Web-приложение по его Web-страницам, а проверка компонента исследует бизнес-уровень *ниже* Web-страниц приложения.

Автоматизируйте проверки системы

Проверки системы (system test) подразумевают исследование всей программной системы, а следовательно, требуют ее полной установки, включая контейнер сервлета и базу данных. При этом проверяется также работоспособность внешних интерфейсов, таких

как Web-страницы, конечные точки Web-служб и графические интерфейсы пользователя. Выполнение проверки системы, как правило, занимает много времени, и разработка тоже. Но когда вы успешно запускаете автоматизированные проверки модулей и компонентов, вы заранее выявляете ряд низкоуровневых проблем и можете достаточно просто спланировать периоды выполнения этой более длительной проверки, возможно, в составе вторичного интеграционного построения, или в определенные часы, или даже ночью.

Проверка системы существенно отличается от проверки функций, она проверяет систему подобно тому, как ее использовал бы клиент. Например, проверка в листинге 6.5 подражает браузеру, манипулируя сайтом при помощи HTTP; но эта проверка *не использует* браузер. Для создания проверки функций применяется такая среда, как Selenium[2], которая управляет браузером. После завершения автоматизированной проверки системы вы вполне можете выполнить автоматизированную или ручную проверку функций, одно другому не мешает.

Листинг 6.5 содержит пример проверки JWebUnit, предпринимающий попытку регистрации на Web-сайте, а затем проверяющий ее успех. Хотя в данном коде это может быть и не очевидно, но вся система (контейнер сервлета и база данных) на момент проверки должна быть установлена и работать. Обратите внимание, что установка здесь не входит в процесс проверки, а является общим аспектом построения.

Листинг 6.5. Проверка системы с использованием JWebUnit

```
public class LoginTest extends WebTestCase {

    protected void setUp() throws Exception {
        getTestContext().
            setBaseUrl("http://pone.acme.com/meinst/");
    }

    public void testLogIn() {
        beginAt("/");
        setFormElement("j_username", "aader");
        setFormElement("j_password", "a1445");
        submit();
        assertTextPresent("Logged in as aader");
    }
}
```

Автоматизируйте проверки функций

Проверка функций (functional test), как и следует из ее названия, подразумевает проверку функциональных возможностей приложения с точки зрения клиента. Это означает, что проверка подражает клиенту. Такие проверки еще называются *приемочными испытаниями* (acceptance test).

Как уже упоминалось, среда, подобная Selenium, фактически управляет браузером и позволяет ему взаимодействовать с Web-сайтом. Проверки среды Selenium пишут в форме таблицы, представляющей полный процесс разработки, включая команды и утверждения. Код листинга 6.6 — это пример проверки Selenium, который предпринимает попытку регистрации на Web-сайте, а затем проверяет ее успешность.

[2] Selenium — это Web-ориентированный, независимый от браузера инструмент проверки функций, доступный на *www.openqa.org/selenium/*.

Листинг 6.6. Проверка функций с использованием Selenium

```
TestLoginSuccess
open                    /ib/app
verifyTitle             Integrate Button - Welcome
verifyTextPresent       Welcome to The IntegrateButton.com.
                        Please log in to access exclusive
                        material for the book
clickAndWait            link=Log In
type                    inputUserId                                  admin
type                    inputPassword                                admin
clickAndWait            loginSubmit
assertTextPresent       Logout
clickAndWait            Link=Logout
assertTextPresent       Log In
verifyTitle             Integrate Button - Welcome
assertTextPresent       Welcome to The IntegrateButton.com.
                        Please log in to access exclusive
                        material for the book
```

Как демонстрирует листинг 6.6, среда Selenium использует для проверки табличную модель, являющуюся настолько высокоэффективным механизмом связи, что автор может и не быть разработчиком. Как можно заметить, эта проверка выполняет ряд действий: проверяет разные аспекты страницы, включая заполнение формы и проверку данных.

Необходимо общее понимание того, что проверки различаются в частности требованиями по установке элементов (подключение к базе данных и т.д.), что непосредственно соотносится с продолжительностью их выполнения. Классификация проверок особенно важна в контексте CI, поскольку продолжительность построения существенно зависит от них и может решительно повлиять на восприятие CI вами и вашей группой.

Категоризируйте проверки разработчика

Запись и выполнение проверок — это, безусловно, хорошее дело, но если рассматривать их как компонент архитектуры, который требует соответствующей структуры и классификации, то они могут начать выглядеть как препятствие, а не как ключ к успеху. Поскольку по мере выполнения проекта объем его базового кода увеличивается, то и количество проверок растет, и если вы будете всегда запускать все написанные проверки в системе CI, то и время построения будет увеличиваться.

Категоризация *проверок разработчика* (developer test) в соответствующую группу (проверки модуля, проверки компонента, проверки системы и даже проверки функций) поможет вам запускать медленные проверки после более быстрых. Например, запуск проверки системы при каждом изменении в хранилище, уведомляющей всех заинтересованных лиц в случае проблем при построении, занимает много времени и ресурсов. Если такая задержка слишком длинна и разработчики успевают перейти к другим действиям, одно из главных преимуществ непрерывной интеграции теряется. Почему бы не запускать проверки модуля при каждом изменении кода, поскольку они не занимают много времени, а далее

по расписанию или через определенные интервалы (или после передающих построений) можно было бы запускать проверки компонентов, а еще реже проверки системы? К концу итерации интервалы можно увеличивать, но на начальных этапах проекта вы, вероятно, хотели бы запускать их чаще.

Такие среды, как NUnit для .NET и версии JUnit и TestNG для Java, имеют аннотации, которые существенно облегчают категоризацию проверок; в других средах сегрегация проверок немного сложней. Например, ни у прежних версий среды JUnit, ни у Ant нет внутреннего механизма, облегчающего разделение проверок на группы. Но все это можно реализовать, воспользовавшись *схемой именования* (naming scheme), или даже еще проще, при помощи *стратегии каталога* (directory strategy).

Одна из практик разработчика подразумевает размещение проверок модуля в отдельном каталоге, а не в том, где находится исходный код. Предположим, например, что в структуре каталога проекта папка `src` предназначена для исходного кода, а для соответствующих проверок — папка `test`. Типичный проект мог бы иметь структуру корневого каталога `root`, подобную представленной в листинге 6.7.

Листинг 6.7. Каталог типового проекта

```
root
    build.xml
    build.properties
    src/
    test/
```

Каталог `src` содержит каталоги, в которых находится исходный код, в то время как каталог `test` включает каталоги, конкретизирующие тип находящихся в них проверок (`unit`, `component` и `system`). Примерное содержимое каталога `test` представлено в листинге 6.8.

Листинг 6.8. Содержимое каталога `test`

```
test/
    unit/
    component/
    system/
```

Каталоги `unit`, `component` и `system` в листинге 6.8 содержат соответствующие проверки для каждой категории. Каталог `system`, например, предназначен для структуры каталогов, связанных с именами пакетов проверок системы (который обычно соответствуют классу в пакете проверки), как показано в листинге 6.9.

Листинг 6.9. Типичная структура каталога `system`

```
test/
    system/
        test/
            com/
                acme/
                    stock/
                        LogInTest.java
                        AccountTest.java
```

Теперь, когда проверки разложены в отдельные каталоги, выбранную вами систему построения следует модифицировать. В случае Ant запуск категоризированных проверок становится вопросом определения целевых объектов, которые используют элемент `batchtest`, находящийся в задаче Ant JUnit, представленной в листинге 6.10.

Листинг 6.10. Элемент `batchtest` задачи JUnit

```
<batchtest todir="${testreportdir}">
    <fileset dir="test/unit">
        <include name="**/*Test.*"/>
            </fileset>
</batchtest>
```

Схема имен, использованная в элементе `include`, общепринята и относится к каталогу в атрибуте `dir` дескриптора `fileset`, определяющего подлежащую запуску проверку, которой в данном случае является проверка модуля.

Не забывайте, что вы можете также автоматизировать проверки функций, например, при помощи системы Selenium; однако они будут следовать иной парадигме выполнения с запуском дополнительных испытаний, которые могут быть легко разделены, например, на индивидуальные задачи Ant. Определяя общий способ категоризации проверок, например при помощи аннотации или схем именования, вы инструктируете систему CI запускать проверки каждой категории в соответствующий момент и полностью управлять временем построения. Это означает, что проверки могут быть запущены через равномерные интервалы, если окажется, что они слишком продолжительны.

Выполняйте более быстрые проверки сначала

Как правило, большая часть времени построения тратится на проверки, и самые длительные из них подразумевают проверку зависимости от внешних объектов, таких как база данных, файловая система и Web-контейнеры. Проверки модулей требуют минимум времени на установку (по определению вообще не требуют), а испытание системы забирает значительное время. Определяя и группируя проверки по типу (модуль, компонент и система), группа разработки может ранжировать процесс построения, при котором проверки запускаются *по категориям*, а не все сразу. Проверки модуля запускаются чаще всего (при каждой передаче); проверки компонента, системы и функций могут быть запущены при последующем построении или по расписанию.

Проверки модуля

На самом деле проверка модуля должна завершаться (успешно) за доли секунды. Если она длится дольше, она заслуживает внимания, поскольку либо нарушена, либо это не проверка модуля, а проверка на уровне компонентов. Помните мантру XP “Небольшая проверка, небольшое изменение кода, небольшая проверка”, в ее основе принцип быстродействия проверки. Если проверка модуля занимает столько времени, что разработчик успевает заняться чем-то еще, значит, она слишком длинна. А это быстро надоедает, и разработчики начинают избегать проверок, а не полагаться на них.

В среде CI построение запускается в любое время, когда некто передает изменения в хранилище с контролем версий. Следовательно, проверки модуля тоже должны запускаться при каждой передаче. Их настройка проста, а ресурсоемкость незначительна.

Проверки компонента

Проверки компонента, которые обычно имеют несколько зависимостей, проходят немного дольше. Независимо от того, выполняются ли они в составе последующих построений или периодически, они должны быть запущены перед передачей кода в хранилище (в закрытом построении). Как упоминалось в главе 4, проверки компонентов могут быть выполнены в составе вторичного и более "тяжеловесного" интеграционного построения, которое следует за передающим построением. Проверки компонента имеют для группы существенное значение: зависимости должны быть на месте и настроены. Эти проверки могут занимать по несколько секунд; но не забывайте, их время складывается. Некоторые проекты с облегченными проверками компонентов могут без проблем выполнять их при каждом передающем построении.

Например, проверка компонента, представленная в листинге 6.11, занимает в среднем четыре секунды.

Листинг 6.11. Пример проверки компонента

```
using System;
using System.Collections;
using NUnit.Framework;
using NHibernate.Cfg;
using NDbUnit.Core.OleDb;
using NDbUnit.Core;

namespace NHibernate.words
{
    [TestFixture]
    public class WordTest
    {
        private const string CONN = @"Provider=SQLOLEDB..";
        private const string SCHEMA = @"Dataset2.xsd";
        private const string XML = @"XMLFile2.xml";

        private OleDbUnitTest fixture;
        private ISessionFactory sessFact;

        [SetUp]
        public void SetUp()
        {
            this.fixture = new OleDbUnitTest(CONN);
            this.fixture.ReadXmlSchema(SCHEMA);
            this.fixture.ReadXml(XML);
            this.sessFact =
               new Configuration().Configure().BuildSessionFactory();
        }

        [Test]
        public void verifyFinder()
        {
            this.fixture.PerformDbOperation(
                        DbOperationFlag.CleanInsert);
```

```
            ISession session = this.sessFact.OpenSession();

            IQuery qry =
                session.GetNamedQuery("word.finder.bySpelling");
            qry.SetAnsiString("spelling", "pugnacious");
            IList list = qry.List();

            Assert.AreEqual(((Word)(list[0])).PartOfSpeech, "adj");
            session.Close();
        }
    }
}
```

Эта проверка делает две вещи, увеличивающие время ее проведения, а также усложняют настройку. Сначала она проверяет подключение к базе данных при помощи среды проверки баз данных NDbUnit[3]. В данном случае NDbUnit вставляет данные из файла `XMLFile2.xml` (это называется также этапом анализа XML). Затем происходит настройка `Nhibernate`, потом проверка его запуска и получение слова из базы данных.

Ничего удивительного, что эта проверка занимает четыре секунды. Каждый дополнительный случай проверки в этом классе добавляет не слишком много времени; но если их наберется с десяток, то общее время приблизится к минуте.

Проверки системы

Проверки системы и функций, для которых требуется полная установка системы, занимают больше всего времени. Кроме того, их полную автоматизацию иногда ограничивает сложность настройки полнофункциональной системы. Запуск проверок системы при каждом передающем построении вполне может стать причиной проблем, поэтому эти типы проверок выполняются при вторичном построении или периодически. Для них хорошо подходит запуск по расписанию (ночью).

Когда в следующий раз будете добавлять случай проверки в построение, учтите продолжительность выполнения всех своих проверок и задумайтесь об оптимизации построения и категоризации, позволяющей организовать их выполнение.

Пишите проверки для дефектов

Проверка разработчика и CI могут *снизить* частоту появления дефектов программного обеспечения, но избежать их *совсем* не удастся. Ошибки, несомненно, случаются, их следует устранять и, в идеале, изучать. Допущение той же ошибки дважды совершенно неприемлемо.

Некоторые используют термин *разработка методом устранения дефектов* (defect-driven development), когда речь идет о создании проверок для дефектов; но данный термин всегда звучал довольно негативно. Не *дефекты* управляют разработкой[4], ею управляет их *предотвращение*! Если какой-нибудь дефект останавливает разработку, его следует устранить, а затем гарантировать, что он не повторится. Для этого и существует стратегия проверки.

[3] NDbUnit — проект с открытым исходным кодом для платформы .NET. Доступно по адресу *www.ndbunit.org/*.

[4] Дословно, defect-driven — управляемый дефектами. — *Примеч. ред.*

Когда дефект проявляется, найдите и изолируйте содержащий его код. Если проект имеет достаточное количество проверок, то дефект, вероятно, находится в непроверяемой части кода (возможно, неправильно указан путь) или кроется во взаимодействии компонентов. Листинг 6.12, например, демонстрирует метод find в классе DAO `Hibernate`, который пытается получить слово из базы данных.

Листинг 6.12. Дефект и DAO

```
public IWord findWord(String word) throws FindException{
    Session sess = null;
    try{
        sess = WordDAOImpl.sessFactory.getHibernateSession();

        final Query qry =
            sess.getNamedQuery("word.finder.bySpelling");
        qry.setString("spelling", word);

        final List lst = qry.list();
        final IWord wrd = (IWord)lst.get(0);
        sess.close();
        return wrd;
    }catch(Throwable thr){
        try{sess.close();}catch(Exception e){}
        throw new FindException("Exception while finding word: "
                + word + " "+ thr.getMessage(), thr);
    }
}
```

Этот класс был достаточно проверен набором проверок на уровне элементов с использованием DbUnit. Эти проверки охватили базовые операции CRUD (create, read, update, delete — создание, чтение, обновление, удаление). Листинг 6.13 демонстрирует пример проверки метода find.

Листинг 6.13. Пример проверки метода

```
public void testFindVerifyDefinition()throws Exception{
    final WordDAOImpl dao = new WordDAOImpl();
    final IWord wrd = dao.findWord("pugnacious");

    for(Iterator iter = wrd.getDefinitions().iterator();
            iter.hasNext();){
        IDefinition def = (IDefinition)iter.next();
        TestCase.assertEquals(
                    "def is Combative in nature; belligerent.",
                    "Combative in nature; belligerent.",
                    def.getDefinition());
    }
}
```

В ходе проверки функций большего приложения (в данном случае, словаря) обнаруживается, что при попытке искать слово, которое в словаре отсутствует, приложение выдает стек трассировки исключения, ошеломляющий пользователя. После некоего расследования оказалось, что если API `Hibernate` не возвратили никакого слова, то метод `findWord`

объекта `WordDAOImpl` передает непредвиденное исключение `IndexOutOfBoundsException` (маскируемое `FindException`).

Такое отклонение в поведении необъяснимо! Но дефект обнаружен! Однако не все потеряно. Помните, мы можем наступать на грабли, но только *один* раз. Мы можем устранить этот сбой, но если он возникнет снова, нам придется *заново* переосмыслить весь подход.

Первым шагом по восстановлению самоуважения станет написание проверки, которая выявляет дефект. Прочитайте это предложение снова и медленно. Вашей первой реакцией, возможно, будет устранение сбойного кода и переход к другому, более захватывающему делу (в добрый час!); однако если вы встанете на этот путь, то потеряете превосходную возможность гарантировать, что та же ошибка никогда не появится снова. Сначала напишите проверку, которая вызывает то же самое поведение, о котором было сообщено в отчете по дефекту. В данном случае необходимо заставить код передать исключение `IndexOutOfBoundsException`, как это сделано в листинге 6.14. Не забудьте, что мы пишем проверку, воспроизводящую поведение, а не проблему.

Листинг 6.14. Проверка на дефект

```
public void testFindInvalidWord() throws Exception{
    final WordDAOImpl dao = new WordDAOImpl();
    try{
        final IWord wrd = dao.findWord("fetit");
        TestCase.fail("This should throw an exception");
    }catch(FindException ex){
        Throwable thr = ex.getOriginalException();
        TestCase.assertTrue("Should be instance of " +
                            IndexOutOfBoundsException",
                            ex.getOriginalException() instanceof
                            IndexOutOfBoundsException);
    }
}
```

Если вы запустите эту проверку, она пройдет. Следовательно, вы доказали существование дефекта. Теперь можете его устранить.

Данная методология, между прочим, немного отличается от преобладающего подхода "разработки методом устранения дефектов", предполагающего написание проверки, которая сначала *не проходит*, но ее следует продолжать запускать (при устранении дефекта), пока она не пройдет успешно. Пример проверки для разработки методом устранения дефектов приведен в листинге 6.15.

Листинг 6.15. Пример проверки для разработки методом устранения дефектов

```
public void testFindInvalidWordException() {
    final WordDAOImpl dao = new WordDAOImpl();
    try{
        final IWord wrd = dao.findWord("fetit");
    }catch (FindException e){
        TestCase.fail("Didn't find word fetit");
    }
}
```

При первом запуске, безусловно, эта проверка потерпит неудачу (подразумевается, что дефект еще не устранен). Такая практика вполне работоспособна; однако ее можно усо-

вершенствовать. Написание проверки, которая преднамеренно терпит неудачу *сначала*, создает следующие проблемы.

- Трудно написать терпящую неудачу проверку, в которой правильно используется метод `assert`.
- Из-за этого метод `assert` не может быть добавлен даже после того, как проверка пройдет успешно. Это означает, что проверка не обязательно сработает даже при наличии дефекта.
- На данном этапе сложно предугадать, как исправления повлияют на поведение, поэтому при попытке провалить проверку вам остается полагаться на удачу.
- В листинге 6.15 сделано предположение, что исправление предотвратит передачу исключения. Это так, но только в данном случае.
- Если исправление было сделано в коде, при проверке терпящая неудачу проверка сработает; однако фактически она не проверяет изменений в поведении.

На настоящий момент, поскольку проверка работает, большинство людей не будут возвращаться к ней для модификации. В данном случае, чтобы устранить дефект, мы в сущности должны провалить проверку, которая в отличие от разработки методом устранения дефектов проходит.

Более внимательное исследование кода указывает на необходимость проверять пустой список перед попыткой получения первого элемента. В настоящий момент мы стоим перед выбором: должен ли код возвратить значение `null`, пустое слово или передать исключение? Предположим, что принято решение возвращать значение `null`, если при помощи Hibernate из базы данных не может быть получено значение параметра (см. листинг 6.16).

Листинг 6.16. Модифицированный код, в котором дефект устранен

```
public IWord findWord(String word) throws FindException{
    Session sess = null;
    try{
        sess = WordDAOImpl.sessFactory.getHibernateSession();

        final Query qry =
            sess.getNamedQuery("word.finder.bySpelling");
        qry.setString("spelling", word);

        final List lst = qry.list();
        IWord wrd = null;
        if(lst.size() > 0){
            wrd = (IWord)lst.get(0);
        }
        sess.close();
        return wrd;
    }catch(Throwable thr){
        try{sess.close();}catch(Exception e){}
        throw new FindException("Exception while finding word: "
                + word + " "+ thr.getMessage(), thr);
    }
}
```

Когда дефект в коде, казалось бы, устранен, проверка запускается снова, и на сей раз терпит неудачу. Следующее решение как раз и отличает данный подход от других. Исправив проверяемый случай, мы утвердим новое поведение. Управляемый дефектом пример работал бы до сих пор с вероятностью пропустить проверяемый случай. Но чего не обеспечивает случай проверки теперь, так это слишком большого количества значений. Нам необходим метод `assert`, возвращающий значение `null`, когда в метод `findWord` передается недопустимое слово. Мы также используем `assert` для предотвращения передачи исключения. Модифицированный случай проверки представлен в листинге 6.17.

Листинг 6.17. Модифицированный случай проверки, контролирующий исправление

```
public void testFindInvalidWord() throws Exception{
    final WordDAOImpl dao = new WordDAOImpl();
    try{
        final IWord wrd = dao.findWord("fetit");
        TestCase.assertNull(
            "Should have received back a null object", wrd);
    }catch(FindException ex){
        TestCase.fail("This should not throw an exception");
    }
}
```

Теперь все закончено, и мы сделали две вещи. Во-первых, исправили дефект. Поздравляем! Во-вторых, установили регрессионную проверку, которая контролирует правильность поведения.

Какой практике следовать: разработке методом устранения дефектов или разработке с непрерывным предотвращением? Обе позволяют следующее:

- устранить дефект;
- предотвратить повторение дефекта.

Однако разработка с непрерывным предотвращением позволяет вам сделать третий шаг, который дает возможность проконтролировать любое новое поведение, вызванное устранением дефекта.

Сделайте проверки компонента воспроизводимыми

Большинство Web-приложений работают с базами данных. Однако база данных — это большое затруднение для проверки, решаемое двумя способами: либо при помощи *ложного объекта* (mock)[5], либо путем максимально долгого избежания использования базы данных в целом, в противном случае приходится платить полную цену и использовать базу данных. Последнее представляет новый ряд проблем — как управлять базой данных в ходе проверки? Или еще лучше: как сделать проверки воспроизводимыми?

Самый простой способ испечь "пирог" проверки *и* съесть его — использовать среду взаимодействия с базой данных, такую как NDbUnit для .NET, DbUnit для Java или PDbSeed для Python. Эти среды абстрагируют набор данных базы в файлы XML, а затем предоставляют разработчику полный контроль над передачей этих данных обратно в базу при проверке. Например, фрагмент кода, представленный в листинге 6.18, взят из файла XML для DbUnit.

[5] Он же заглушка. — *Примеч. ред.*

Листинг 6.18. Пример файла данных DbUnit

```
<word WORD_ID="1" SPELLING="pugnacious" PART_OF_SPEECH="Adjective"/>
<definition DEFINITION_ID="10"
    DEFINITION="Combative in nature; belligerent."
    WORD_ID="1"
    EXAMPLE_SENTENCE=
        "Thepugnacious youth had no friends left to pick on."/>
<synonym SYNONYM_ID="20" WORD_ID="1" SPELLING="belligerent"/>
<synonym SYNONYM_ID="21" WORD_ID="1" SPELLING="aggressive"/>
```

При помощи класса DbUnit `DatabaseTestCase` данные из файла XML можно вставлять, модифицировать и удалять. Для настройки определенной базы данных реализуют абстрактный метод `getConnection` и находят файл XML при помощи метода `getDataSet` (листинг 6.19).

Листинг 6.19. Пример проверки базы данных

```
public class DefaultWordDAOImplTest extends DatabaseTestCase {
    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSet(
            new File("test/conf/words-seed.xml"));
    }

    protected IDatabaseConnection getConnection() throws Exception {
        final Class driverClass =
              Class.forName("org.gjt.mm.mysql.Driver");

        final Connection jdbcConnection =
        DriverManager.getConnection(
            "jdbc:mysql://localhost/words",
            "words", "words");
        return new DatabaseConnection(jdbcConnection);
    }
    public void testFindVerifyDefinition() throws Exception{
        final WordDAOImpl dao = new WordDAOImpl();
        final IWord wrd = dao.findWord("pugnacious");

        for(Iterator iter =
                wrd.getDefinitions().iterator(); iter.hasNext();){
            IDefinition def = (IDefinition)iter.next();
            assertEquals("Combative in nature; belligerent.",
                         "Combative in nature; belligerent.",
                         def.getDefinition());
        }
    }

    public DefaultWordDAOImplTest(String name) {
        super(name);
    }
}
```

Обратите внимание, этот класс, тем не менее, предполагает, что база данных находится на той же машине, на которой выполняется проверка. На рабочей станции разработчика данное предположение вполне может оправдаться, но в средах CI такая конфигурация, безусловно, может превратиться в проблему.

Одно из решений заключается в отказе от жестко заданной строки подключения и помещения ее в файл свойств. Существует, однако, и более эффективный механизм. Если для подключения к базе данных применяется DbUnit, то вы можете использовать это приложение непосредственно. Если дело обстоит так, что применяется общепринятая практика (избежать жесткого задания информации подключения внутри базового кода), то почему бы не настроить среду DbUnit на чтение того же самого файла, который приложение читает при проверке?

Например, в приложениях Hibernate информация для подключения к базе данных обычно определяется в файле `hibernate.cfg.xml`. Вы можете легко написать вспомогательный класс, который анализирует этот файл и получает соответствующую информацию подключения. Но еще лучше, как продемонстрировано в листинге 6.20, для предоставления желаемой информации полагаться на Hibernate.

Листинг 6.20. Утилита настройки Hibernate

```
public class DBUnitHibernateConfigurator {
    static Configuration configuration = null;

    private DBUnitHibernateConfigurator() {
        super();
 }

private static Configuration getConfiguration()
    throws HibernateException {
        if (configuration == null) {
            configuration = new Configuration().configure();
        }
        return configuration;
}

public static IDataSet getDataSet(final String fileName)
    throws ResourceNotFoundException,
    DBUnitHibernateConfigurationException {
    try{
        return DBUnitConfigurator.getDataSet(fileName);
    }catch(DBUnitConfigurationException e2){
        throw new DBUnitHibernateConfigurationException(
            "DBUnitConfigurationException in getDataSet", e2);
    }
}

private static String getProperty(final String name)
    throws HibernateException {
        return getConfiguration().getProperty(name);
    }

public static Properties getHibernateProperties()
```

```
    throws ResourceNotFoundException,
    DBUnitHibernateConfigurationException{
        try{
            final Properties hProp = new Properties();
            hProp.put("hibernate.connection.driver_class",
                DBUnitHibernateConfigurator.getProperty(
                    "hibernate.connection.driver_class"));
            hProp.put("hibernate.connection.url",
                DBUnitHibernateConfigurator.getProperty(
                    "hibernate.connection.url"));
            hProp.put("hibernate.connection.username",
                DBUnitHibernateConfigurator.getProperty(
                    "hibernate.connection.username"));
            hProp.put("hibernate.connection.password",
                DBUnitHibernateConfigurator.getProperty(
                    "hibernate.connection.password"));
            return hProp;
        }catch(HibernateException e){
            throw new DBUnitHibernateConfigurationException(
            "HibernateException in getHibernatePropertiesFile", e);
        }
    }
}

public static IDatabaseConnection getDBUnitConnection()
    throws DBUnitHibernateConfigurationException{
        try{
            final Properties props =
                DBUnitHibernateConfigurator.getHibernateProperties();
            return DBUnitConfigurator.getDBUnitConnection(props);
        }catch(DBUnitConfigurationException e1){
            throw new DBUnitHibernateConfigurationException(
                "DBUnitConfigurationException in
                getDBUnitConnection", e1);
        }catch (ResourceNotFoundException e2) {
            throw new DBUnitHibernateConfigurationException(
                  "ResourceNotFoundException in getDBUnitConnection",
                  e2);
        }
    }
}
```

Обратите внимание, как класс в листинге 6.20 помещает информацию подключения Hibernate в объекте `Properties`, который затем преобразуется в тип DbUnit `IDatabaseConnection` в классе `DBUnitConfigurator`. Тип подключения DbUnit возвращает метод `getDBUnitConnection`. Тип DbUnit `IDataSet`, предоставляющий файлы XML, которые содержат все данные, возвращает метод `getDataSet`. Этот метод освобождает разработчиков от необходимости указывать путь к файлу, что особенно затруднительно для разных сред.

В листинге 6.21 создается специальный абстрактный класс испытания, который запрашивает у реализатора (implementer) информацию, необходимую для специфического случая проверки.

Листинг 6.21. Подходящий случай проверки

```
public abstract class DefaultDBUnitHibernateTestCase extends
DatabaseTestCase {
    public DefaultDBUnitHibernateTestCase(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
        DefaultHibernateSessionFactory.
            closeSessionAndEvictCache();
        DefaultHibernateSessionFactory.
            getInstance().getHibernateSession();
    }

    protected void tearDown() throws Exception {
        DefaultHibernateSessionFactory.
            closeSessionAndEvictCache();
        super.tearDown();
    }

    protected IDatabaseConnection getConnection() throws Exception {
        return DBUnitHibernateConfigurator.
            getDBUnitConnection();
    }

    protected IDataSet getDataSet() throws Exception {
        final String fileName = this.getDBUnitDataSetFileForSetUp();
        DatabaseTestCase.assertNotNull("data set file was null",
            fileName);
        return DBUnitHibernateConfigurator.getDataSet(fileName);
    }

    protected abstract String getDBUnitDataSetFileForSetUp();
}
```

Пример полученного в результате случая проверки, реализующего класс `DefaultDBUnitHibernateTestCase`, представлен в листинге 6.22.

Листинг 6.22. Новый случай проверки в действии

```
public class WordDAOImplTest extends DefaultDBUnitHibernateTestCase {

    public void testUpdateWordSpelling()throws Exception{
        WordDAOImpl dao = new WordDAOImpl();
        IWord wrd = dao.findWord("pugnacious");

        wrd.setSpelling("pugnacious-ness");
```

```
        dao.updateWord(wrd);

        IWord wrd2 = dao.findWord("pugnacious-ness");
        assertEquals("should be id of 1", 1, wrd2.getId());
    }

    public void testFindVerifyDefinitionsSize() throws Exception{
        WordDAOImpl dao = new WordDAOImpl();
        IWord wrd = dao.findWord("pugnacious");

        Set defs = wrd.getDefinitions();
        assertEquals("size should be one", 1, defs.size());
    }

    protected String getDBUnitDataSetFileForSetUp() {
        return "words-seed.xml";
    }

    public WordDAOImplTest(String name) {
        super(name);
    }
}
```

DbUnit предоставляет API (как упомянуто ранее), которые можно эффективно использовать совместно, что обеспечивает огромные возможности в комбинаторных средах. Обладая такой дополнительной гибкостью, проверка различных архитектур на разных уровнях становится очень простой. Например, проверка разработчика в приложении Struts может быть сложной. Общепринятая тактика подразумевает использование среды типа HttpUnit, моделирующей запросы HTTP; но это может оказаться утомительной работой, причем не обеспечивающей желаемой точности для архитектуры Struts, которая часто использует классы `Action` и конфигурацию для связывания запросов.

Проект StrutsTestCase был создан для решения данной проблемы. При помощи этой среды вы можете легко изолировать и проверять классы Struts `Action`. Но этот проект требует, чтобы разработчик дополнял базовый класс, который обрабатывает ложный контейнер сервлета. Если приложение Struts вынуждает использовать базу данных, вы можете оказаться в затруднительном положении.

При помощи API DbUnit может быть создана комбинаторная среда, использующая возможности подключения DbUnit совместно с возможностями ложного объекта проекта StrutsTestCase (листинг 6.23).

Листинг 6.23. Комбинация проверки Struts и Hibernate

```
public abstract class DefaultDBUnitMockStrutsTestCase
    extends MockStrutsTestCase {

    public DefaultDBUnitMockStrutsTestCase(String testName) {
        super(testName);
    }

    public void setUp() throws Exception {
        super.setUp();
```

```
        this.executeOperation(this.getSetUpOperation());
    }

    public void tearDown() throws Exception{
        super.tearDown();
        this.executeOperation(this.getTearDownOperation());
    }

    private void executeOperation(DatabaseOperation operation)
    throws Exception{
        if (operation != DatabaseOperation.NONE){
                final IDatabaseConnection connection =
                        this.getConnection();
            try{
                operation.execute(connection, this.getDataSet());
            }finally{
                closeConnection(connection);
            }
        }
    }

    protected void closeConnection(IDatabaseConnection connection)
    throws Exception{
        connection.close();
    }
    protected abstract Properties getConnectionProperties();

    protected abstract String getDBUnitDataSetFileForSetUp();

    protected IDatabaseConnection getConnection() throws Exception {
        final Properties dbPrps = this.getConnectionProperties();
        DatabaseTestCase.
            assertNotNull("database properties were null", dbPrps);
        return DBUnitConfigurator.getDBUnitConnection(dbPrps);
    }

    protected DatabaseOperation getSetUpOperation() throws Exception
    {
        return DatabaseOperation.CLEAN_INSERT;
    }

    protected DatabaseOperation getTearDownOperation()
    throws Exception {
        return DatabaseOperation.NONE;
    }

    protected IDataSet getDataSet() throws Exception {
        final String fileName = this.getDBUnitDataSetFileForSetUp();
        DatabaseTestCase.assertNotNull("dataset file was null",
            fileName);
        return DBUnitConfigurator.getDataSet(fileName);
    }
}
```

Вы можете еще раз остановиться на жестко заданной информации подключения или многократно использовать для этого уже существующие файлы. Возможна ли проверка приложения Struts, применяющего Hibernate? Без проблем, достаточно объединить новый класс `DefaultDBUnitMockStrutsTestCase` со вспомогательной утилитой для чтения файлов Hibernate.

В листинге 6.24 приведен пример класса, реализующего класс `DefaultMerlinMockStrutsTestCase`, который объединяет возможность класса DbUnit `DefaultDBUnitMockStrutsTestCase` со вспомогательной утилитой чтения файлов Hibernate, определенной ранее в листинге 6.20.

Листинг 6.24. Комбинированная среда в действии

```
public class ProjectListActionTest
        extends DefaultMerlinMockStrutsTestCase {

    public void testProjectListAction() throws Exception{
        this.setRequestPathInfo("/viewProjects");
        this.actionPerform();
        this.verifyForward("success");

        IProject[] projects = (IProject[])this.getRequest().
                    getAttribute("projects");
        assertNotNull("object was null", projects);
    }
    public ProjectListActionTest(String name) {
        super(name);
    }

    protected String getDBUnitDataSetFileForSetUp() {
        return "dbunit-project-seed.xml";
    }
}
```

Теперь вы имеете превосходный пример проверки, который можно предъявить любому, кто жалуется на невозможность организовать проверку такого приложения воспроизводимым способом.

Ограничьте проверку одним методом `assert`

При плотном графике разработки, особенно когда надвигается "счастливое" время сдачи, весьма соблазнительно попытаться объединить все варианты в одну проверку. Это ведет к изобилию методов `assert` в одном случае проверки. Например, код листинга 6.25 попытается в одном варианте проверки контролировать поведение метода `buildHierarchy` класса `HierarchyBuilder`, а также поведение объекта `Hierarchy`.

Листинг 6.25. Случай проверки со слишком многими методами `assert`

```
public void testBuildHierarchy()throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
        "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be 2", 2,
        hier.getHierarchyClassNames().length);
```

```
    assertEquals("should be junit.framework.TestCase",
        "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
    assertEquals("should be junit.framework.Assert",
        "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
}
```

Обратите внимание на три метода `assert` в листинге 6.25. Это вполне допустимый случай проверки JUnit; ничто не запрещает использовать при этом разные методы `assert`. Однако проблема данной практики заключается в том, что JUnit осуществляет построение с *быстрым отказом* (fast-failing). Если сбой обнаружит первый метод `assert`, то случай проверки сообщит об этом сразу и остановит дальнейшую проверку. Это означает, что следующие два метода `assert` не будут запущены и не выполнят проверку.

Как только код будет исправлен, а проверка запущена снова, неудачу может потерпеть второй метод `assert`, что по новой запустит весь цикл исправления и следующей проверки. Если при следующей попытке сбой обнаружит третий метод `assert`, то все придется повторить снова. Вы обратили внимание на неэффективность такой схемы?

Более эффективной является практика *ограничения каждого случая проверки одним методом* `assert`. Данный способ быстрее, чем повторение процесса в три этапа, как было описано только что, поскольку вы можете получить уведомление обо всех отказах в результате *одной* проверки. Например, код листинга 6.25 может быть рефакторингован в три отдельных набора проверок (листинг 6.26).

Листинг 6.26. Рефакторинг случая проверки

```
public final void testBuildHierarchyStrSize() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
        "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
     assertEquals("should be 2", 2,
         hier.getHierarchyClassNames().length);
}

public final void testBuildHierarchyStrNameAgain() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
        "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.TestCase",
        "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
}

public final void testBuildHierarchyStrName() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
        "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.Assert",
        "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
}
```

Имея три отдельных случая проверки, вы получите сообщения о трех отказах при первом запуске проверки. Таким образом, вы можете ограничиться одним циклом исправ-

ления. Безусловно, данная практика ведет к быстрому увеличению количества проверок. Вот почему в начале этой главы была описана отдельная структура каталога для проверок. Причем количество проверок растет пропорционально размеру вашего кода!

Резюме

Насколько надежным вы хотите сделать ваше программное обеспечение? Исходный код надежен настолько, насколько он покрыт проверками, которые выполняются достаточно часто. Сегрегация проверок на четыре автономные категории (модуля, компонента, системы и функций) позволяет настроить систему CI так, чтобы она выполняла проверки наиболее эффективным способом. Проверки модуля могут быть запущены при каждом изменении, а проверки компонентов системы и функций — при последующем построении, например, по расписанию.

Табл. 6.1 подводит итог практик, описанных в этой главе.

Таблица 6.1. Практики CI, обсуждаемые в этой главе

Практика	Описание
Автоматизируйте проверки модуля	Автоматизируйте проверки модуля предпочтительно с использованием среды проверки модулей, такой как NUnit или JUnit. Проверки модуля не должны иметь никаких внешних зависимостей, таких как файловая система или база данных
Автоматизируйте проверки компонента	Автоматизируйте проверки компонентов с применением среды проверки модулей, такой как JUnit, NUnit, DbUnit или NDbUnit, если вы используете базу данных. Эти проверки подразумевают большее количество объектов и обычно требуют намного больше времени, чем проверки модуля
Автоматизируйте проверки системы	Проверки системы проходят дольше, чем проверки компонента, и обычно включают несколько компонентов
Автоматизируйте проверки функций	Проверки функций могут быть автоматизированы с использованием таких инструментов, как Selenium (для Web-приложений) и Abbot (для приложений GUI). Проверки функций работают с точки зрения пользователя и обычно длятся дольше всех остальных проверок в автоматизированном проверочном комплекте
Категоризируйте проверки разработчика	Распределяя проверки по категориям, вы можете запускать длительные проверки (например, компонентов) отдельно от более быстрых (например, модуля)
Выполняйте более быстрые проверки сначала	Запускайте проверки модуля до проверок компонентов, системы и функций. Это достигается за счет категоризации проверок
Пишите проверки для дефектов	Увеличьте покрытие кода за счет написания проверок для обнаруженных дефектов, что гарантирует также оповещение об их повторном проявлении
Сделайте проверки компонента воспроизводимыми	Используйте среду проверки базы данных, чтобы удостовериться в корректности их состояния, что поможет сделать проверки компонента воспроизводимыми
Ограничьте проверку одним методом `assert`	Ограничивая автоматизированные проверки одним методом `assert`, вы тратите меньше времени на отслеживание всех причин отказа проверки

Вопросы

Используйте приведенный список вопросов, чтобы оценить процесс проверки в свете среды CI и того, что она может вам дать.

- Категоризируете ли вы свои автоматизированные проверки, например, на проверки модуля, компонента, системы и функций?
- Настраиваете ли вы систему CI так, чтобы запускать каждую категорию проверок на различных стадиях построения?
- Пишете ли вы автоматизированные проверки модуля для каждого дефекта?
- Сколько методов `assert` находятся в каждом вашем случае проверки? Ограничиваетесь ли вы одним методом `assert` для каждого случая проверки?
- Все ли ваши проверки автоматизированы? Автоматизирована ли в вашем проекте передача проверок от разработчика в хранилище с контролем версий?

Глава 7

Непрерывная инспекция

Велик тот человек, который способен использовать умственные способности других людей для решения своих вопросов.

Донн Пиатт (Donn Piatt)

Просмотр кода двумя людьми, как правило, положительно сказывается на его качестве в целом, поскольку данный подход предоставляет дополнительную пару глаз для анализа. По той же причине практика парного программирования рекомендована в XP. Такие инструменты статического анализа исходного кода, как PMD для Java и FxCop для .NET, позволяют просматривать файлы на предмет нарушения заранее установленных правил, обеспечивая ряд преимуществ при анализе.

Однако все три метода анализа кода (обзор кода, парное программирование и статический анализ кода) окажутся не слишком полезными при построении, если не применять их регулярно. Кроме того, обзоры кода и парное программирование осуществляются людьми, которые склонны к ошибкам, не могут обеспечить высокую пропускную способность и быстро устают от бесконечно повторяемых задач.

Серьезный подход к анализу кода, например при помощи славного инспекционного процесса Fagan[1], *может* быть весьма эффективен; но все же он выполняется людьми, а люди, как известно, склонны к эмоциям. Это означает, что сотрудники могут забыть сообщить коллегам о своих сомнениях относительно кода, к тому же участники одного коллектива имеют тенденцию *субъективно* оценивать работу своих коллег. Учтите также цену времени, растрачиваемого на обзоры кода, даже с использованием наиболее эффективных сред.

[1] Более подробная информация об инспекционном процессе Fagan приведена по адресу `http://en.wikipedia.org/wiki/Fagan_inspection`.

Как уже отмечалось, парное программирование эффективно тогда, когда оно применяется правильно. Наличие второй пары глаз, постоянно просматривающей код, поможет создать более высококачественный код; однако организационные вопросы такой столь творческой методики являются недостатком. На пары разработчиков, кстати, также могут распространяться те же проблемы, т.е. эмоции и субъективность.

Между инспекцией человеком и инструментом статического анализа существует два различия.

- Эти инструменты невероятно дешевы при частом запуске. Они требуют человеческого вмешательства только при настройке и первом запуске, после которого они работают автоматически и обеспечивают экономию на почасовой ставке сотрудников.
- Данные инструменты полагаются на неустрашимую и неумолимую объективность компьютера. Последний не станет идти на компромиссы типа "Ваш код выглядит прекрасно, если вы скажете то же о моем коде", а также не будет проситься по нужде и на перерыв, когда вы запускаете автоматизированный инструмент инспекции при каждом изменении содержимого хранилища с контролем версий.

Эти инструменты также можно настраивать — организация может выбирать наиболее подходящие правила для своего базового кода и проверять его соответствие им *каждый раз* при передаче кода в хранилище с контролем версий. Эти инструменты становятся, по сути, неустанными контролерами исходного кода, подражать которым человеку фактически невозможно.

Эти инструментальные средства также очень хорошо работают в географически распределенных группах (т.е. одни разработчики работают дома, другие в офисе, а остальные по разным странам, континентам и т.д.), что помогает снизить любые дополнительные риски, связанные с человеческим фактором.

В случае больших объемов базового кода его автоматизированный статический анализ существенно эффективней, чем просмотр человеком; некоторые инструменты предоставляют сотни разных правил, которые человеку проблематично запомнить и использовать при просмотре множества файлов. Кроме того, применение инструментом к базовому коду бесчисленного количества правил займет гораздо меньше времени, чем просмотр *одного пакета* вашим партнером. Не следует также забывать, что просмотр *всего* кода человеком стоит довольно дорого!

Автоматизация инспекции кода при помощи инструментальных средств анализа позволяет выполнять порядка 80% работ, оставляя людям лишь 20% наиболее ответственного кода. Например, PMD для Java применит к файлу более 180 правил *при каждом* его изменении. При нарушении особенно важного условия, например превышено значение цикломатической сложности[2], кто-то может и не обратить на это внимания. Попробуйте вообразить планирование такого процесса, осуществляемого вручную. А выполнить его не хотите? Главное, что следует уяснить по поводу автоматизированного обзора кода, — это то, что он не заменит человеческих рук. Это просто усиление человеческого интеллекта, который собственно и необходим.

Мы не пропагандируем сценарий "или, или", при котором вы должны решить, какую именно методику инспекции применять — автоматизированную или ручную. Инструменты автоматизированной инспекции предпочтительней персональных, их необходимость возросла потому, что код стал значительно объемней и сложней. Преимущество автоматизированной инспекции кода заключается в том, что она куда эффективней инспекции вруч-

[2] *Цикломатическая сложность* (cyclomatic complexity) — это количество возможных путей в разделе кода, например в методе. Более подробно обсуждается далее в этой главе.— *Примеч. ред.*

ную, поскольку замечает низкоуровневые детали кода. Человеку лучше сосредоточиться на тех аспектах, которые автоматизировали инструменты, например, на соответствии кода требованиям заказчика и предстоит ли его поддерживать в будущем.

Рис. 7.1 демонстрирует, что инспекция — это еще один неотъемлемый атрибут процесса построения системой CI, запускаемого одной командой.

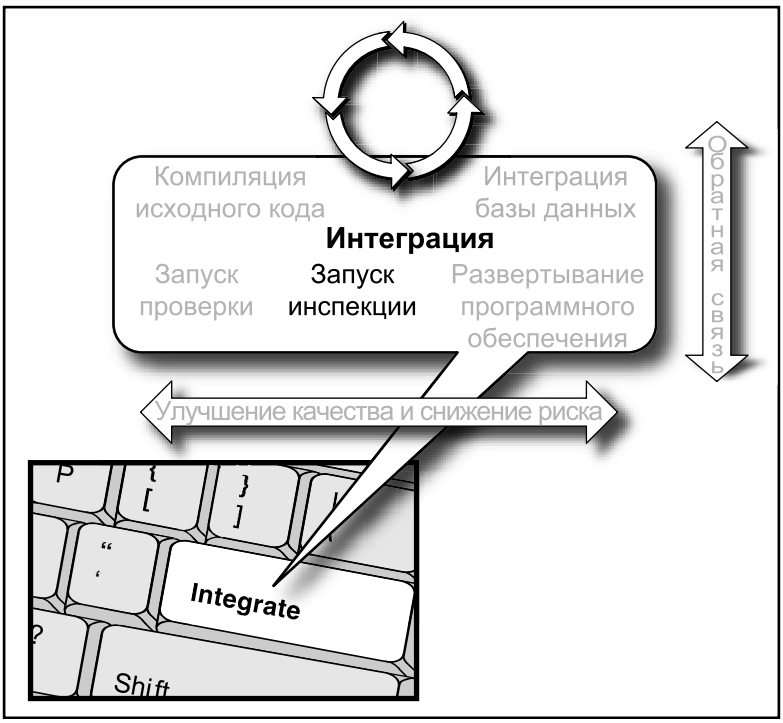


Рис. 7.1. Кнопка <Integrate> — запуск инспекции

В чем разница между инспекцией и проверкой?

Между проверкой и инспекцией программного обеспечения есть различие. *Проверка* осуществляется динамически и подразумевает исследование функциональных возможностей программного обеспечения. *Инспекция* — это анализ кода на основании набора предопределенных правил. В главе 6, “Непрерывная проверка”, описывалось несколько типов проверок программного обеспечения, включая проверку модуля, компонента и системы. *Инспекторы* (inspector) — это инструментальные средства статического или динамического анализа, которые контролируют соответствие принятым для группы стандартам (как правило, неким показателям программирования или проектирования). Инспекции подлежат, например, стандарты “грамматики” кода, соблюдение архитектурных уровней, дублирование кода и многое другое, что мы обсудим в данной главе. Проверка и инспекция — это похожие концепции, в том смысле, что они не изменяют код программного обеспечения, а лишь демонстрируют, где могут находиться проблемы. Безусловно, одних инспекций и проверок недостаточно для повышения качества программного обеспечения; в их ходе необходимо самому принимать меры для решения выявленных проблем.

Как часто следует осуществлять инспекцию?

Непрерывная инспекция уменьшает промежуток времени между обнаружением проблемы и ее устранением. Вы также экономите время сотрудников для собственно устранения проблем. Инспекции программного обеспечения позволяют выявить в системе области, заслуживающие повышенного внимания. В действительности те разработчики, которые работают вручную, способны проинспектировать лишь небольшие области системы. Но как вы определите, какие из областей исследовать и в каком состоянии они находятся на данный момент? Вопрос даже не в том, *сможете* ли вы найти дефект, а *когда* вы его найдете, ведь исправить дефект проще, пока еще помнишь логику, предположения и подробности данной области. После этого компоненты программного обеспечения следует просмотреть снова.

В проектах, где обзоры выполняются только вручную, проблема в коде может скрываться несколько месяцев, прежде чем будет обнаружена. В результате время потеряно, контекст проблемы, возможно, тоже. Но если процесс написания кода сопровождается немедленным запуском автоматизированных инспекторов (а также проверок), вы обеспечите себе спокойное будущее, где дефекты будут обнаруживаться и, вероятно, устраняться за несколько минут. Уменьшение промежутка между проявлением дефекта и его устранением улучшает качество кода. Безусловно, предотвращение дефектов предпочтительней их устранения, и инспекции оказывают в этом неоценимую помощь.

Находите дефекты прежде, чем они проявятся

Уменьшайте период между обнаружением дефекта и его устранением при помощи непрерывной инспекции.

Многие IDE обладают встроенными средствами инспекции, способными помочь автоматизировать написание кода за счет указания неиспользуемых переменных и случаев лексической некорректности (например, совпадения имен). Весьма поощряется применение IDE для локального запуска автоматизированной инспекции, но эти же инспекции следует запускать и при автоматизированном построении, а также в ходе CI, чтобы предотвратить ложный положительный отзыв и гарантировать воспроизводимость и однозначность подхода.

Показатели кода: история

Десятилетие назад для оценки кода на предмет потенциальных дефектов требовались услуги нескольких интеллектуальных сотрудников. Это было интересное занятие: исследовать шаблоны сбойного кода в надежде, что создание и использование формальной модели поможет обнаружить проблемы *прежде, чем они станут дефектами*. При правильном применении это обеспечивало полезнейшую информацию для усовершенствования кода.

Затем некие другие умные люди задались вопросом: а нельзя ли, используя код, измерять производительность труда разработчика. На первый взгляд это казалось достаточно честным: “Дэвид написал больше кода, чем Билл; следовательно, его работа производительней и заслуживает большего вознаграждения. Кроме того, я заметил, что Билл часто болтается без дела. Я полагаю, нам следует расстаться с ним”. Вполне очевидно, что этим показателем можно злоупотреблять. Некоторые счетчики количества строк кода учитывают комментарии. Кроме того, данный показатель фактически одобряет разработку в стиле копирования и вставки. Впоследствии оказывается: “Дэвид допускал много дефек-

тов! Каждый второй дефект, который мы находим, создан им. Это плохо, что мы уволили Билла, его код практически не содержал дефектов".

Классический показатель — количество строк кода на разработчика — оказался на редкость обманчив[3]. Это, возможно, удивит большинство менеджеров, но не разработчиков. К счастью, маятник качнулся обратно, и люди пришли к мнению, что следует учитывать также сложность и безотказность кода.

Снижайте сложность кода

Вы, несомненно, обращали внимание на то, что длинные методы зачастую трудно исследовать. Вам приходилось испытывать проблемы при понимании логики в чрезмерно глубоко вложенных условных выражениях? Ваши инстинкты вас не обманули. Большие методы и методы с высоким количеством путей выполнения трудно понять, а кроме того, их сложность, фактически, прямо пропорциональна вероятности возникновения дефектов.

Многочисленные исследования показали явную корреляцию между количеством путей выполнения кода и вероятностью дефектов. Один из показателей, разработанных в результате этих исследований, называется *числом цикломатической сложности* (Cyclomatic Complexity Number — CCN). CCN — это просто целое число, которое определяет сложность по количеству возможных путей выполнения метода. За последние годы различные исследования с использованием данного показателя позволили установить, что методы с числом CCN выше 10 имеют более высокий риск дефектов, чем другой код того же объема[4].

JavaNCSS для Java[5] — это превосходный инструмент определения длины методов и классов в ходе исследования файлов исходного кода. Он также подсчитывает цикломатическую сложность каждого метода в базовом коде. Настраивая JavaNCSS при помощи соответствующей задачи Ant или дополнения Maven, можно получать отчеты XML, содержащие следующие данные:

- количество классов, методов, некомментированных строк кода и изменение стилей комментариев в каждом пакете;
- количество некомментированных строк кода, методов, внутренних классов и комментариев Javadoc в каждом классе;
- общее количество некомментированных строк кода и значение цикломатической сложности.

Инструмент JavaNCSS связан с несколькими таблицами стилей, которые позволяют создавать отчет HTML, содержащий необходимые данные. Рис. 7.2 демонстрирует пример отчета HTML, созданного системой Maven.

Этот раздел отчета, озаглавленный "Top 30 functions containing the most NCSS" ("30 функций, содержащих наибольшие значения NCSS"), содержит подробности о самых больших методах в базовом коде, что обычно совпадает с высокой цикломатической сложностью. Например, в отчете указано, что метод `findAllStates` класса `BeerDaoImpl` насчитывает 238 строк кода и обладает цикломатической сложностью (CCN) 114.

---

[3] См. *www.martinfowler.com/bliki/CannotMeasureProductivity.html*.

[4] См. *www.sei.cmu.edu/str/descriptions/cyclomatic_body.html*.

[5] JavaNCSS доступен по адресу *www.kclee.de/clemens/java/javancss/*. Значения CCN для платформы .NET предоставляют CCMetrics и Source Monitor.

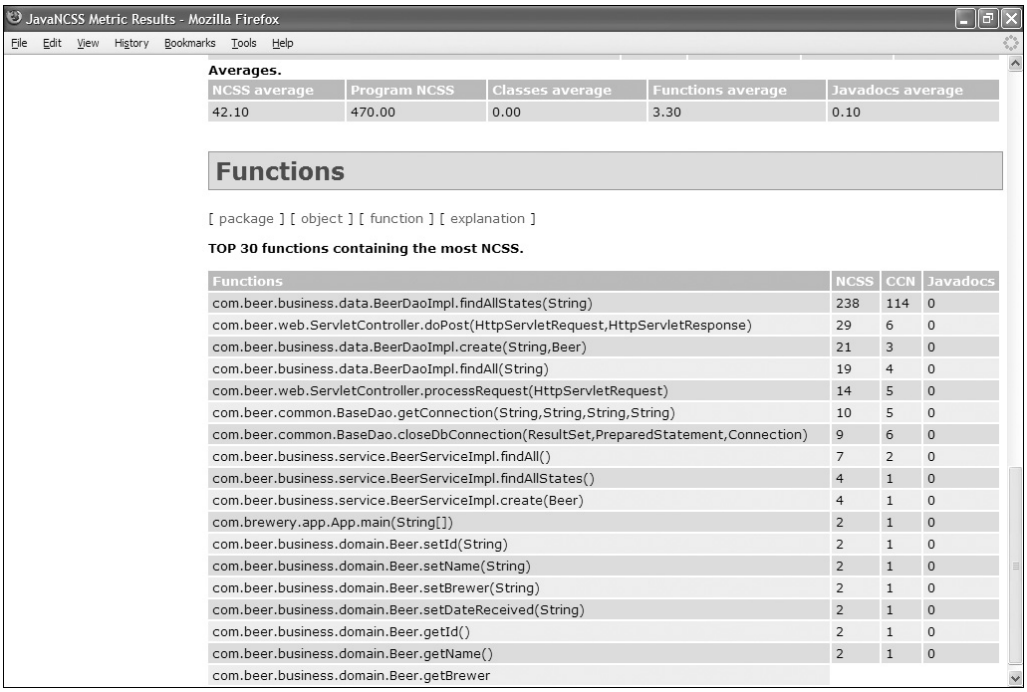


Рис. 7.2. Отчет CCN, созданный системой Maven

У вас, возможно, возник вопрос: “Ну и что это значит?”

Поскольку высокие значение цикломатической сложности свидетельствует о склонности к дефектам, нам необходимо предпринять следующие действия. Если проверки имеются, то сколько их? Эмпирическое правило для покрытия проверками относительно цикломатической сложности гласит, что количество случаев проверки должно быть равно значению цикломатической сложности (т.е. для метода `findAllStates` в примере понадобилось бы 114 проверок). Маловероятно, что в реальности для этого метода действительно проводилось бы 114 проверок, но увеличение их количества — уже прекрасное начало для снижения риска возникновения дефектов.

Если для данного метода нет соответствующего набора проверок, то рискованный участок налицо, и вам необходимо немедленно написать адекватные проверки. Некоторые могут подумать, что как раз настал момент для рефакторинга; но это нарушило бы его первое правило: прежде чем что-либо изменять, напишите случай проверки[6]. Как только набор проверок будет готов, вы можете приступать к снижению риска, идя путем рефакторинга. Наиболее эффективный способ снижения цикломатической сложности заключается в применении подхода извлечения метода[7] и распределения сложности на меньшие, а следовательно, более управляемые и более проверяемые методы. Безусловно, следующим шагом после создания меньших методов является написание инспекций и проверок для них.

В среде CI оценка изменения сложности метода *по времени* становится возможной. Получив вначале отчет об инспекции данного метода, вы можете контролировать рост

[6] См. раздел **The Value of Self-testing Code** главы 4 книги Мартина Фаулера *Refactoring*.

[7] См. *www.refactoring.com/catalog/extractMethod.html*.

(или снижение) значения сложности при последующих инспекциях. Если наблюдается рост, можно предпринять соответствующие действия.

Если значение CCN метода продолжает возрастать, то группа может предпринять следующее:

- удостовериться, что метод имеет достаточное количество проверок для снижения риска;
- оценить возможность рефакторинга метода, чтобы уменьшить вероятные проблемы при последующем обслуживании.

Поскольку JavaNCSS также способен документировать тенденции, его отчеты можно использовать для контроля организационных стандартов. Инструмент сообщает об одиночных и многострочных комментариях, подходящих для дополнения Javadocs. В некоторых кругах программистов считается, что наличие большого количества комментариев — это свидетельство относительной сложности кода.

JavaNCSS — не единственный инструмент, способный облегчить создание отчетов о сложности на платформе Java. PMD — еще один проект с открытым исходным кодом, который анализирует файлы исходного кода Java, руководствуется рядом правил создания отчета о сложности, включая замер цикломатической сложности, размера классов и методов. Checkstyle — еще один проект с открытым исходным кодом и подобными правилами. Инструменты PMD и Checkstyle, как и JavaNCSS, обладают соответствующими задачами Ant и дополнениями Maven.

Как уже упоминалось, сложность коррелирует с дефектами. Используйте инспекции для контроля значения сложности базового кода и тенденций его изменения, а также принимайте адекватные меры (создание проверок и рефакторинг) для снижения риска возникновения дефектов.

Осуществляйте обзоры проекта непрерывно

В конце XX столетия появились и другие полезные показатели. Вы когда-нибудь замечали, что объекты, имеющие много зависимостей от других объектов, не обладают высокой надежностью? При внесении изменения в одну из зависимостей и сам объект может быть нарушен. И наоборот, когда вы изменяете объект, то другие объекты в системе, зависимые от него, могут создать проблемы в другом месте. (Это явление обычно называют эффектом "сопутствующего нарушения".) Во избежание непредвиденного следует соблюдать меру в изменениях, а чтобы себя не ограничивать, по возможности избегайте зависимостей.

Два показателя, наиболее полезные в определении "связей", называются *центростремительная связь* (Afferent Coupling) и *центробежная связь* (Efferent Coupling), а иногда именуются как *коэффициент разветвления по входу* (Fan In) и *коэффициент разветвления по выходу* (Fan Out) соответственно). Эти целочисленные показатели подсчитывают количество входящих и исходящих связей объекта. Показатели центростремительной и центробежной связи выражают архитектурную сложность обслуживания: несет ли объект ответственность перед слишком многими другими объектами (высокое значение центростремительной связи) или объект очень сильно зависит от других объектов (высокое значение центробежной связи).

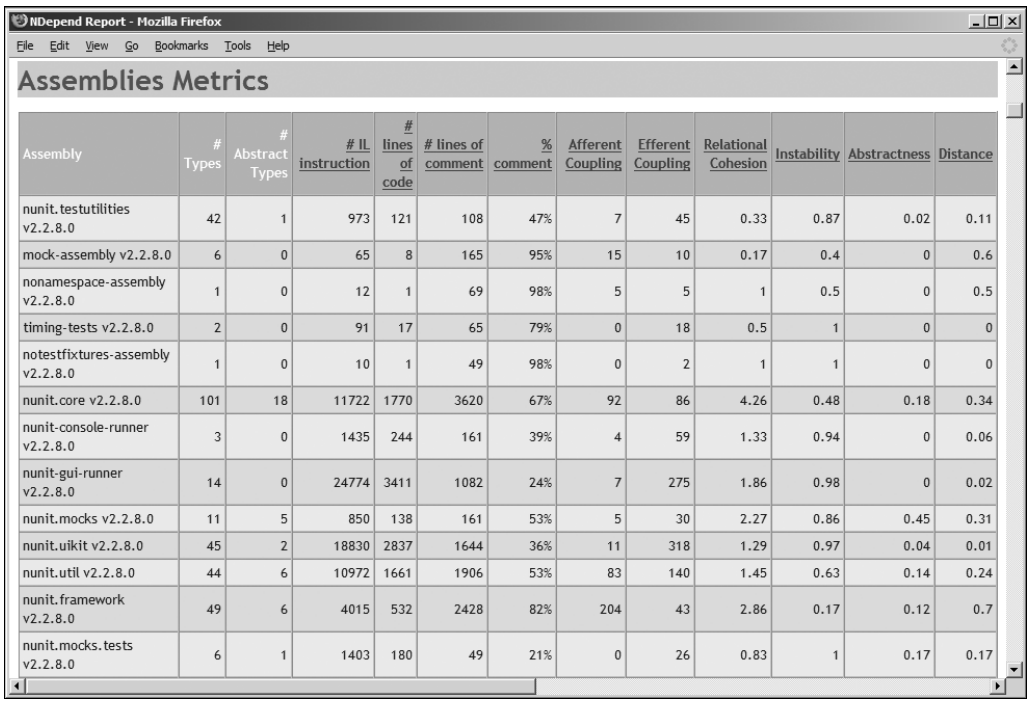
Данные показатели зависимости могут быть чрезвычайно полезны при определении риска при поддержке базового кода. Объекты, пространства имен или пакеты со слишком большой ответственностью создают большой риск, подвергаясь изменениям. Изменение их поведения так или иначе скажется на других объектах в программной системе, которые,

в свою очередь, могут прекратить функционировать правильно. Объекты, которые сильно зависят от других объектов, уязвимы перед лицом изменений в них и также могут перестать правильно работать.

Но что важнее всего: критерии центростремительной и центробежной связи могут быть объединены в показатель *неустойчивости* (Instability). Например, приведенное ниже уравнение поможет вычислить уровень неустойчивости объекта (или пространства имен или пакета) перед лицом изменений. Обратите внимание, что значение 1 соответствует абсолютной неустойчивости, а значение 0 — полной устойчивости.

Неустойчивость = центробежная связь/(центробежная связь + центростремительная связь)

NDepend для платформы .NET — это проект с открытым исходным кодом, который создает отчеты об уровне центробежной и центростремительной связи, неустойчивости и ряде других интересных архитектурных показателей, которые представляются по сборкам и классам. Инструмент легко запускается при помощи NAnt и создает отчеты в форматах XML или HTML. Пример отчета HTML о показателях для сборки .NET представлен на рис. 7.3 (в данном случае это базовый код среды NUnit).



Assemblies Metrics

Assembly	# Types	# Abstract Types	# IL instruction	# lines of code	# lines of comment	% comment	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
nunit.testutilities v2.2.8.0	42	1	973	121	108	47%	7	45	0.33	0.87	0.02	0.11
mock-assembly v2.2.8.0	6	0	65	8	165	95%	15	10	0.17	0.4	0	0.6
nonamespace-assembly v2.2.8.0	1	0	12	1	69	98%	5	5	1	0.5	0	0.5
timing-tests v2.2.8.0	2	0	91	17	65	79%	0	18	0.5	1	0	0
notestfixtures-assembly v2.2.8.0	1	0	10	1	49	98%	0	2	1	1	0	0
nunit.core v2.2.8.0	101	18	11722	1770	3620	67%	92	86	4.26	0.48	0.18	0.34
nunit-console-runner v2.2.8.0	3	0	1435	244	161	39%	4	59	1.33	0.94	0	0.06
nunit-gui-runner v2.2.8.0	14	0	24774	3411	1082	24%	7	275	1.86	0.98	0	0.02
nunit.mocks v2.2.8.0	11	5	850	138	161	53%	5	30	2.27	0.86	0.45	0.31
nunit.uikit v2.2.8.0	45	2	18830	2837	1644	36%	11	318	1.29	0.97	0.04	0.01
nunit.util v2.2.8.0	44	6	10972	1661	1906	53%	83	140	1.45	0.63	0.14	0.24
nunit.framework v2.2.8.0	49	6	4015	532	2428	82%	204	43	2.86	0.17	0.12	0.7
nunit.mocks.tests v2.2.8.0	6	1	1403	180	49	21%	0	26	0.83	1	0.17	0.17

Рис. 7.3. Отчет NDepend

Обратите внимание, что сборка `nunit.framework` имеет значение центростремительной связи (Afferent Coupling) 204 и значение центробежной связи (Efferent Coupling) 43. Это означает, что изменение данного кода (базовый код среды NUnit) может вызвать трудности. Следовательно, значение неустойчивости (Instability) для данной сборки равно 0.17. Это значит, что от него зависит множество других объектов, а также существует мизерная вероятность того, что изменение данного кода *не повлияет* на другие объекты.

О другой сборке, `nunit.mocks.tests`, содержащей проверки, инструмент NDepend сообщил, что ее значение центробежной связи составляет 26, а значение центростремительной связи — 0. Следовательно, значение неустойчивости равно 1, т.е. код абсолютно неустойчив. Это имеет следующий смысл: любые изменения кода приводят, как правило, к неудаче проверки (если это не так, то с проверками проблемы).

Учет данных показателей может оказать драматическое воздействие на ремонтопригодность базового кода. Например, сборки с высоким значением центростремительной связи должны быть снабжены бо́льшим количеством проверок, если вы хотите гарантировать надежность зависимого кода. Долгосрочный учет значения центростремительной связи может подвигнуть группу к решению разделить сборки на меньшие, более гибкие фрагменты кода.

В то время как высокие значения центростремительной связи присущи объектам, которые вызывают сбои, сборки с высоким значением центробежной связи сами склонны к таковым. Таким образом, достаточный объем покрытия кода проверками этих сборок поможет группе быстрее выявлять проблемы. В среде CI, контролирующей данные значения регулярно, группа разработки может вовремя обнаружить риски и вмешаться прежде, чем дело дойдет до потери контроля. Если вы наблюдаете неуклонные тенденции роста показателей связи, ваша группа может предпринять следующее:

- выявив риски, сразу же создать проверки на их основании;
- постоянно учитывать все факторы уязвимости, обусловленные высокими значениями показателей связи;
- после выполнения проверок провести некий рефакторинг, позволяющий сгладить проблемы в будущем.

Подобно NDepend для платформы .NET, JDepend — это проект с открытым исходным кодом для платформы Java, оповещающий о показателях связи пакетов. JDepend может быть запущен при помощи Ant или Maven и создает отчеты в форматах XML и HTML.

Определяя количество связей сборок, пакетов или объектов, архитектурные показатели связи могут действительно выявлять долгосрочные проблемы обслуживания базового кода. Эти показатели помогут также определить степень риска, связанного с изменениями. Но что важнее всего, регулярно контролируя эти показатели в среде CI, можно действительно снижать риски *прежде*, чем они превратятся в настоящий кошмар обслуживания.

Поддерживайте организационные стандарты при проверке кода

Стандарты программирования облегчают общее понимание базового кода участниками разных групп разработчиков. Подобно тому как в значительной степени был стандартизирован рынок обслуживания автомобилей, чтобы вы могли купить, например, новую лампочку для фары у своего изготовителя или у другого стороннего производителя и заменить ее в любом автосервисе, “структура” базового кода тоже может быть стандартизована, что позволит разным людям быстро оценивать его поведение и изменять при необходимости. Это повысит вашу реакцию при разработке и сделает вас независимым от конкретного разработчика или группы при внесении изменений.

Как уже упоминалось, хотя просмотр кода человеком и парное программирование могут быть эффективны при контроле стандартизованных программ, эти методы не подлежат масштабированию в отличие от автоматизированных инструментов. Последние не

только содержат сотни правил (которые обычно можно настраивать), но и позволяют их часто применять, к тому же без вмешательства человека.

В среде CI инструмент анализа кода может быть запущен *в любое время*, когда в хранилище проекта произошло изменение. Инструмент может анализировать отдельный файл при его изменении или проанализировать весь базовый код, если изменения коснулись структуры или других систем. Но что замечательнее всего, в связи с характером CI заинтересованные лица могут быть немедленно уведомлены о нарушениях в архитектуре или коде. Например, популярный инструмент анализа кода для платформы Java, PMD, насчитывает более 180 настраиваемых правил в категориях, от размещения фигурных скобок в условных выражениях до соблюдения соглашения об именовании, соглашения проектирования (например, упрощения условных выражений) и даже неиспользуемого кода. В Java, если условное выражение имеет только один оператор, фигурные скобки необязательны. Код листинга 7.1, например, полностью допустим в Java. Однако некоторые организации находят такой код опасным, поскольку впоследствии при добавлении дополнительных операторов можно забыть поставить фигурные скобки.

Листинг 7.1. Пример условного выражения без фигурных скобок

```
if(status)
    commit();
```

Код листинга 7.2 вполне допустим, но он имеет замаскированный дефект, который может подстеречь ничего не подозревающего разработчика. Кто бы мог подумать, что команда `commit` будет выполнена, даже если переменная `status` содержит значение `true`? Более того, она окажется реализованной независимо от значения переменной `status`. Инструмент PMD с соответствующим набором правил находит код, который потенциально способен приводить к подобным ошибкам, и сообщает о них в отчете.

Листинг 7.2. Пример условного выражения с логическим дефектом

```
if(status)
    log.debug("committing db");
    commit();
```

Соглашения об именовании — это, как правило, первые аспекты кода, которые определяют группы. Так, например, неописательные, короткие имена переменных и методов могут существенно затруднить понимание их назначения (особенно если их автор больше не работает в компании). Например, метод, представленный в листинге 7.3, вполне мог бы иметь лучшее имя, а также переменные с более описательными именами, чем `s` и `t` (их тип можно выяснить в верхней части метода, но если бы они имели более осмысленные имена, то делать это не понадобилось бы).

Листинг 7.3. Плохо поименованный метод с неописательными переменными

```
public void cw(IWord wrd) throws CreateException {
    Session s = null;
    Transaction t = null;
    try{
        s = WordDAOImpl.sessFactory.getHibernateSession();
        t = s.beginTransaction();
        s.saveOrUpdateCopy(wrd);

        t.commit();
```

```
        s.flush();
        s.close();
    }catch(Throwable thr){
        thr.printStackTrace();
        try{s.close();}catch(Exception e){}
        try{t.rollback();}catch(Exception e){}
        throw new CreateException(thr.getMessage());
    }
}
```

Здесь PMD снова приходит на выручку. Его запуск для этого кода сообщил бы о нарушении правил как для имени метода, так и для односимвольных переменных. По умолчанию для PMD установлен просмотр 3[8]; но группа вправе изменить это значение на большее.

PMD позволяет также упростить код. Например, метод, представленный в листинге 7.4, будучи вполне синтаксически корректным, выглядит чересчур подробным.

Листинг 7.4. Вполне допустимый код, но весьма подробный

```
public boolean validateAddress(){
    if(this.getToAddress() != null){
        return true;
    }else{
        return false;
    }
}
```

Данный метод, помеченный PMD, вполне может быть упрощен, как представлено в листинге 7.5.

Листинг 7.5. Метод, упрощенный благодаря PMD

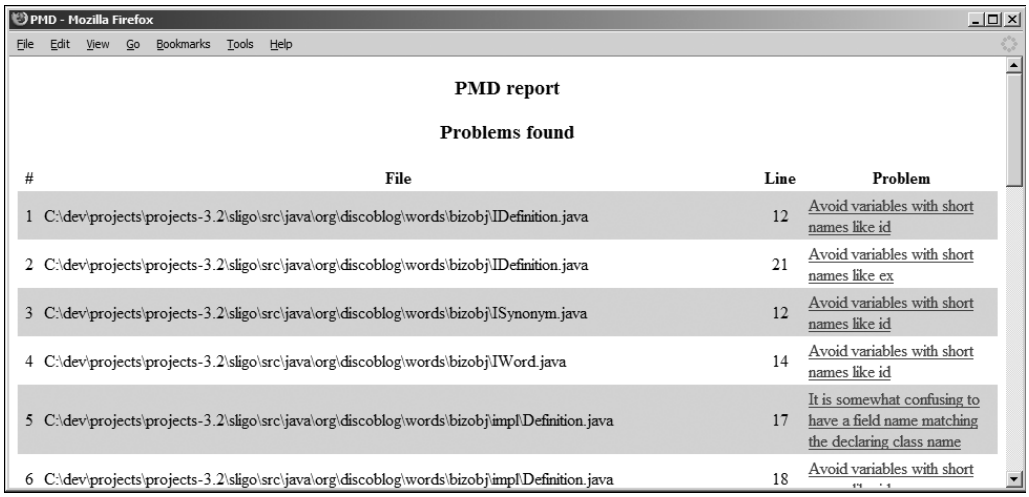
```
public boolean validateAddress(){
    return (this.getToAddress() != null);
}
```

PMD, подобно большинству других инструментариев на рынке, запускается при помощи Ant или Maven и создает отчеты в формате XML, который может быть преобразован в формат HTML. Например, отчет на рис. 7.4 оповещает о нарушениях в ряде файлов `.java` базового кода.

Как уже упоминалось, PMD также может документировать показатели сложности, включая цикломатическую сложность, длину методов и классов. Checkstyle — еще один инструмент с открытым исходным кодом, доступный разработчикам Java и имеющий обширную документацию. Он запускается при помощи Ant или Maven и создает отчеты HTML. FxCop — подобный инструмент для платформы .NET с многочисленными правилами и возможностями создания отчетов. PyLint предназначен для языка Python.

Непрерывно контролируя и инспектируя код, ваша группа сможет постоянно оставаться в рамках архитектурных и программных правил. Раннее выявление дефектов позволит избежать долгосрочных проблем в обслуживании.

[8] Вероятно, имеется в виду длина имени переменной в символах. — *Примеч. ред.*


PMD - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

PMD report

Problems found

#	File	Line	Problem
1	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\IDefinition.java	12	Avoid variables with short names like id
2	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\IDefinition.java	21	Avoid variables with short names like ex
3	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\ISynonym.java	12	Avoid variables with short names like id
4	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\IWord.java	14	Avoid variables with short names like id
5	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\impl\Definition.java	17	It is somewhat confusing to have a field name matching the declaring class name
6	C:\dev\projects\projects-3.2\sligo\src\java\org\discoblog\words\bizobj\impl\Definition.java	18	Avoid variables with short

Рис. 7.4. Отчет PMD

Снижайте количество двойного кода

Разработчики зачастую предпочитают скопировать и вставить код, нежели искать лучшие способы обобщения, многократного использования или абстрагирования поведения. Проблема дублирования кода возникла при написании первых программ, поэтому исследователи и разработчики упорно трудились над ее устранением много лет. Усовершенствование конструкции программ, например введение процедурного программирования, объектно-ориентированного программирования и, впоследствии, аспект-ориентированного программирования помогло уменьшить потребность в дублировании кода. Однако соблазн скопировать и вставить будет существовать всегда. К сожалению, разработчик зачастую просто не понимает, что он создает проблему на будущее.

Копирование и вставка кода могут происходить в любых областях системы в той или иной форме, включая следующие.

- Логика базы данных, в том числе хранимые процедуры и представления, например SQL.
- Компилируемый исходный код, например Java, C, C++ и C#.
- Интерпретируемый исходный код, например ASP, JSP, JavaScript и Ruby.
- Сценарии построения, например make, и файлы построения Ant.
- Файлы данных и конфигурации, например ASCII, XML, XSD и DTD.

Майкл Тумим (Michael Toomim), Эндрю Бегель (Andrew Begel) и Сьюзен Л. Грехэм (Susan L. Graham)[9] отметили следующее: “По результатам недавних исследований ядро Linux (на 2002 год) содержит от 15 до 25 % дублированного кода”[10], а также: “Код JDK Java

[9] См. “Managing Duplicated Code with Linked Editing,” на `http://harmonia.cs.berkeley.edu/papers/toomim-linked-editing.pdf`.

[10] Антониол Г. (G. Antoniol), Пента М. Д. (M. D. Penta), Мерло Е. (E. Merlo) и Виллано У. (U. Villano) “Analyzing cloning evolution in the Linux kernel”, опубликована *Journal of Information and Software Technology* 44(13):755–765, 2002.

от Sun дублирован на 21–29 %"[11]. Дублирование кода — это насущная проблема даже для наиболее популярных программных пакетов, используемых в промышленности[12].

Дублирование кода приводит к следующим проблемам:

- увеличению издержек на поддержку в связи с обнаружением, оповещением, анализом и многократным устранением ошибок;
- неопределенности относительно существования других ошибок (которые не были еще обнаружены в копиях кода);
- увеличению затрат на проверку за счет написания ее дополнительных случаев.

Использование PMD-CPD

Для поиска двойного кода пригодно несколько инструментов. PMD предоставляет *детектор копирования и вставки* (Copy/Paste Detector — CPD) для языков C/C++, Java, PHP и Ruby. Инструмент работает довольно хорошо, прост в установке и применении и может создавать отчеты в виде XML, CSV или текста (ASCII). Листинг 7.6 демонстрирует использование задачи Ant `CPD`.

Листинг 7.6. Использование задачи Ant `CPD`

```
1    <property name="reports.pmd.dir"
             value="${reports.dir}/pmd-reports" />
2    <property name="reports.cpd.dir" value="${reports.pmd.dir}" />
3    <property name="cpd.output.type" value="text"
             description="csv,xml,text"/>
4    <property name="cpd.output.filename"
             value="cpd-results.${cpd.output.type}" />
5    <property name="cpd.output.dir" value="${build.dir}" />
6    <property name="cpd.outputfile"
             value="${cpd.output.dir}/${cpd.output.filename}" />
7    <target name="run-cpd">
8        <taskdef name="cpd"
             classname="net.sourceforge.pmd.cpd.CPDTask"
             classpathref="pmd.classpath" />
9        <cpd minimumTokenCount="20"
             outputFile="${cpd.outputfile}"
             format="${cpd.output.type}"
             ignoreLiterals="true"
             ignoreIdentifiers="true">
10               <fileset dir="${src.dir}">
11                   <patternset refid="non.test.sources.pattern" />
12               </fileset>
13       </cpd>
14   </target>
```

[11] Камия Т. (T. Kamiya), Кусумото С. (S. Kusumoto) и Иноуэ К. (K. Inoue) "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", опубликована *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[12] См. "Managing Duplicated Code with Linked Editing," на `http://harmonia.cs.berkeley.edu/papers/toomim-linked-editing.pdf`.

- **Строка 2**. Назначение для каталога отчетов CPD той же папки, где хранятся отчеты PMD.
- **Строка 3**. В данном примере будет создан текстовый отчет. Вы можете также создать отчет, разделяемый запятыми, или отчет XML.
- **Строка 9**. Вызов задачи `cpd`. Атрибут `minimumTokenCount` определяет количество совпадающих маркеров, чтобы код считался дублированным. Атрибут `ignoreLiterals="true"` заставляет CPD игнорировать строковые литералы при оценке двойного блока. Аналогично, атрибут `ignoreIdentifiers="true"` заставляет игнорировать строковые литералы при оценке идентификаторов (имена переменных, методов).
- **Строки 10–11**. Задают исходный код, подлежащий проверке на дублирование.

Использование Simian

Simian — это еще один инструмент поиска скопированного и вставленного кода. Simian работает с платформой .NET версии 1.1 и далее, языком Java версии 1.4 и далее. Листинг 7.7 демонстрирует, как использовать Simian в Ant.

Листинг 7.7. Использование Simian в задаче Ant

```
1    <property name="reports.simian.dir"
         value="${reports.dir}/simian-reports"/>
2    <property name="simian.output.filename"
         value="simian-results.xml"/>
3    <property name="simian.output.dir" value="${build.dir}"/>
4    <property name="simian.outputfile"
         value="${simian.output.dir}/${simian.output.filename}"/>
5    <path id="simian.classpath">
6        <pathelement location="${lib.dir}/simian-2.2.17.jar"/>
7    </path
8    <target name="run-simian">
9        <delete dir="${reports.simian.dir}"quiet="true"/>
10       <mkdir dir="${reports.simian.dir}"/>
11       <taskdef resource="simiantask.properties"
             classpathref="simian.classpath"/>
12       <simian threshold="4" language="java">
13           <fileset dir="${src.dir}" >
14               <include name="**/*.java"/>
15               <exclude name="**/*Test*"/>
16           </fileset>
17           <formatter type="xml" toFile="${simian.outputfile}"/>
18       </simian>
19   </target>
```

- **Строка 1**. Определяет свойства для расположения отчета Simian о дублировании.
- **Строки 2–4**. Определяют свойства `simian.outputfile`. Выходной файл XML будет помещен в каталог `build`.
- **Строки 5–7**. Создание пути класса Simian для загрузки задачи Ant `simian`.

- **Строки 9–10**. Удаление всех предыдущих отчетов и подготовка к новому отчету о дублировании.
- **Строка 11**. Загрузка задачи Ant `simian`.
- **Строка 12**. Вызов задачи Ant `simian` с указанием языка проверяемого на дублирование кода, в данном случае `java`. Атрибут `threshold` задает минимальное количество строк, совпадение которых считается дублированием.
- **Строки 13–16**. Включить исходный код проекта; исключить любой код проверки.

Simian поставляется с таблицей стилей XSLT, позволяющей преобразовать отчет XML в формат HTML (листинг 7.8).

Листинг 7.8. Создание отчета HTML для Simian

```
1    <available property="simian.outputfile.present"
         file="${simian.outputfile}"/>
2    <target name="simian-report" if="simian.outputfile.present">
3        <xslt
             in="${simian.outputfile}"
             out="${reports.simian.dir}/Simian-Report.html"
             style="${config.dir}/simian/simian.xsl"/>
4    </target>
```

- **Строка 1**. Проверка существования выходного файла Simian (XML) и установка свойства `simian.outputfile.present`, если это так.
- **Строка 2**. Запуск целевого объекта `simian-report`, если свойство `sim-ian.outputfile.present` установлено.
- **Строка 3**. Создание отчета Simian с использованием таблицы стилей XSLT, поставляемой в комплекте Simian.

На рис. 7.5 приведен отчет, который Ant и Simian создали для кода листинга 7.7. Обратите внимание, что код в этом примере имеет дублирование 4.27% с учетом порогового значения 4 строки.

Оценивайте покрытие кода

Существуют разные типы показателей размеров покрытия, но большинство инструментов сосредотачивается на покрытии строк (что еще называется *операторным покрытием*). Покрытие строки означает, что определенная строка кода проверяется. Вы получаете метрики покрытия проверками, запуская базовый код с проверкой и фиксируя данные, которые соответствует коду, затронутому в ходе процесса проверки. Затем из этих данных синтезируется отчет о покрытии. Для проверок кода на языке Java обычно используется JUnit и такие инструменты покрытия, как Cobertura, EMMA или Clover. На платформе .NET может быть применена среда проверки NUnit и такой инструмент покрытия кода, как NCover или Clover.NET.

Например, если длина метода десять строк, семь из которых подвергаются проверке, то он имеет покрытие в 70% строк. Процесс работает также на агрегатном уровне: если класс имеет 100 строк и 65 из них проверяются, то его покрытие составляет 65% строк. Аналогично, если базовый код содержит 10 000 строк некомментированного кода и 3 000 из них подвергаются проверке, то покрытие базового кода составит 30% строк.

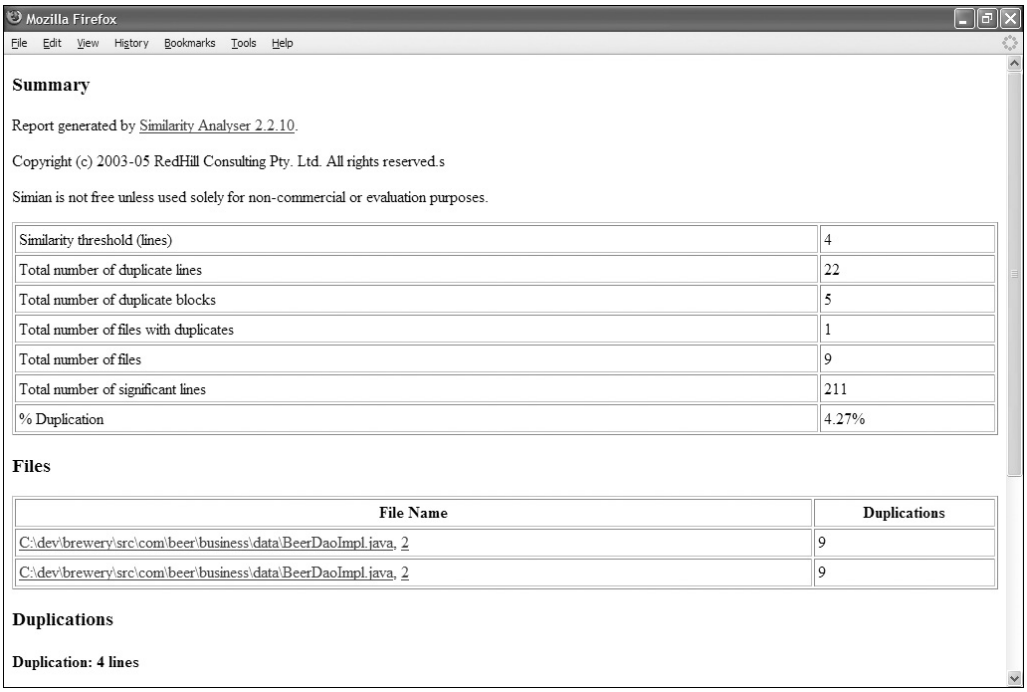


Рис. 7.5. Отчет о дублировании кода, созданный Simian и Ant

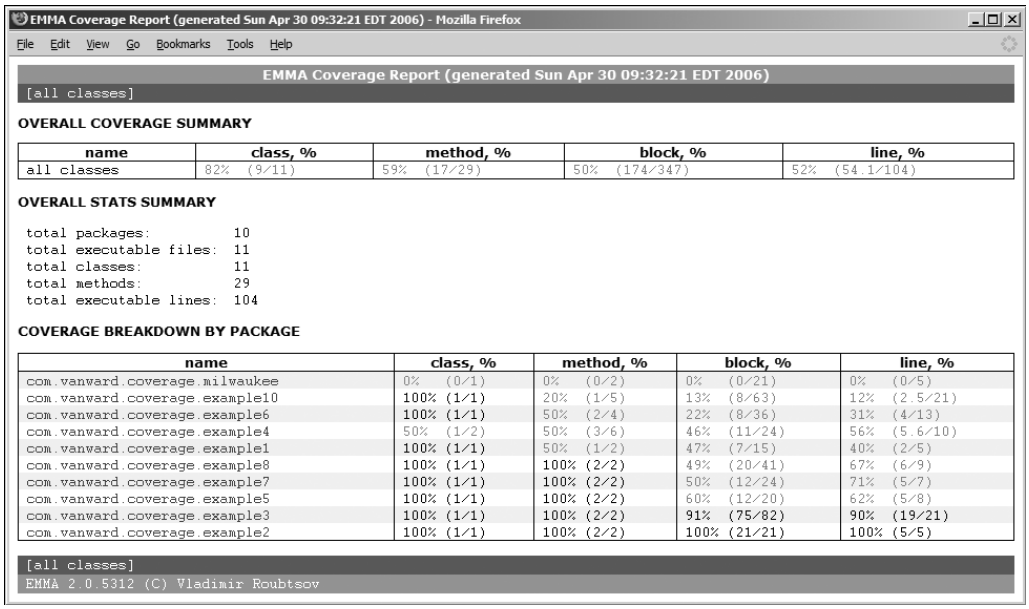
Некоторые инструменты предоставляют также отчет о покрытии переходов (иногда называемом покрытием пути). Они пытаются замерить покрытие точек ветвления, например блоков `if` и `else`. Подобно отчету о покрытии строк, если в определенном методе есть две ветви выполнения и обе покрыты проверками, то можно сказать, что метод имеет покрытие переходов на 100%.

Например, в среде Maven запуск EMMA требует двух шагов. Сначала вы загружаете дополнение и помещаете его в каталог дополнений Maven. Затем запускаете задачу `emma`, которая автоматически компилирует и исходный код, и код проверки. Затем EMMA исследует исходный код и запускает код проверки непосредственно при помощи задачи `test:test`. EMMA создает отчет HTML, подобный представленному на рис. 7.6.

Регулярно оценивайте качество кода

Теперь важнейший аспект: как применять эти показатели? Вы, безусловно, должны использовать инструменты оценки покрытия как часть процесса проверки в среде CI, однако не стоит *переоценивать* информацию, которую они могут вам предоставить. Не забывайте, что отчеты о покрытии лучше всего использовать для выявления кода, который не был адекватно проверен. Когда вы исследуете отчет о покрытии, ищите наиболее низкие значения и выясняйте, почему данный код не был проверен полностью.

Среда QA также может использовать данную информацию для точной настройки проверки функций. Зная, что некоторые разделы базового кода недостаточно покрыты проверками, среда QA может брать части приложения и сосредотачивать свои усилия на подозрительных областях.



EMMA Coverage Report (generated Sun Apr 30 09:32:21 EDT 2006) - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

EMMA Coverage Report (generated Sun Apr 30 09:32:21 EDT 2006)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	82% (9/11)	59% (17/29)	50% (174/347)	52% (54.1/104)

OVERALL STATS SUMMARY

total packages:	10
total executable files:	11
total classes:	11
total methods:	29
total executable lines:	104

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
com.vanward.coverage.milwaukee	0% (0/1)	0% (0/2)	0% (0/21)	0% (0/5)
com.vanward.coverage.example10	100% (1/1)	20% (1/5)	13% (8/63)	12% (2.5/21)
com.vanward.coverage.example6	100% (1/1)	50% (2/4)	22% (8/36)	31% (4/13)
com.vanward.coverage.example4	50% (1/2)	50% (3/6)	46% (11/24)	56% (5.6/10)
com.vanward.coverage.example1	100% (1/1)	50% (1/2)	47% (7/15)	40% (2/5)
com.vanward.coverage.example8	100% (1/1)	100% (2/2)	49% (20/41)	67% (6/9)
com.vanward.coverage.example7	100% (1/1)	100% (2/2)	50% (12/24)	71% (5/7)
com.vanward.coverage.example5	100% (1/1)	100% (2/2)	60% (12/20)	62% (5/8)
com.vanward.coverage.example3	100% (1/1)	100% (2/2)	91% (75/82)	90% (19/21)
com.vanward.coverage.example2	100% (1/1)	100% (2/2)	100% (21/21)	100% (5/5)

[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Рис. 7.6. Отчет покрытия EMMA для CruiseControl

Зная об этом, группы разработки и QA могут использовать инструменты покрытия проверками в среде CI, чтобы избежать ручной проверки функций.

Проверка разработчика снижает *риск* наличия дефектов в коде; поэтому некоторые группы разработчиков требуют теперь писать проверки модуля вместе с вновь создаваемым или изменяемым кодом. CI позволяет гарантировать регулярное выполнение таких проверок в течение разработки, поскольку CI будет запускать их при каждом изменении.

Мониторинг покрытия позволяет группе разработки оперативно выявлять и код, размер которого растет *быстрее* соответствующего количества проверок. Например, получив отчет о покрытии в начале недели, вы узнаете, что ключевой пакет проекта имеет коэффициент покрытия 70%. Если позднее, к концу недели, окажется, что покрытие пакета составляет уже 60%, то вы можете предположить следующее.

- Количество строк кода пакета выросло, но соответствующих проверок для нового кода создано не было (или что вновь добавленные проверки фактически не покрывают новый код).
- Некоторые из проверок были удалены.

Регулярный просмотр отчета упрощает отслеживание прогресса в выполнении задачи, судя по коэффициенту покрытия и соотношению количества проверок на количество строк кода. Если вы замечаете, что проверок пишется недостаточно, вы можете как-то повлиять на разработчиков, например, послав их на обучение, отчитав или применив парное программирование.

Преимуществом данного подхода является очевидность тенденций. Когда сроки поджимают и график уплотняется, люди вполне могут отойти от принципов качества. Куда лучше просматривать информационные отчеты, чем потом разводить руками перед об-

наружившим дефект клиентом: “У нас такое произошло впервые” (а ведь данный дефект можно было обнаружить при простой проверке месяц назад). Да и гнева начальства это позволит вам избежать.

Частота покрытия

Поскольку большинство инструментов покрытия кода обладают дополнительными возможностями для специфических целей (т.е. код имеет “обработчики”, которые оповещают о времени выполнения), проверки с их использованием выполняются медленнее, чем без них. Это может отрицательно сказаться на проверках покрытия в среде CI, если их процесс продуман не очень хорошо. Поэтому запуск инструментов покрытия кода может стать наиболее подходящим в ходе более продолжительных вторичных построений.

Если в группе принято несколько стратегий выполнения проверок на разных этапах рабочего процесса (что связано с категоризацией проверок), то имеет смысл создать дополнительную стратегию для процесса покрытия, чтобы запускать ее один раз в день в составе проверки более продолжительной категории. Например, проверки модуля осуществляются при каждом изменении в хранилище. Проверки компонентов обычно выполняют при последующем построении или регулярно, через определенные промежутки времени в течение дня, а проверки системы проводятся один раз в день (как правило, вечером). После выполнения проверки системы можно запустить также другой набор проверок, в состав которого входит и проверка покрытия (т.е. выполняются проверки модулей, затем проверки компонентов, после чего проверки системы). В результате данного процесса создается ряд отчетов, которые группа может просмотреть на следующее утро.

Имейте в виду, что три разных отчета, сделанных в разной конфигурации, могут указать разную степень покрытия для одного и того же кода. Например, класс `Foo` может иметь покрытие 0% в отчете проверки модуля и высокое покрытие в отчете проверки системы. Кроме того, поскольку будет создано три отчета о покрытии, вы должны настроить процесс построения так, чтобы отчеты не перезаписывали друг друга. Не забывайте перемещать отчеты в соответствующие папки или уникально их переименовывать. Некоторые инструменты, такие как Cobertura для Java, способны объединять отчеты в один главный отчет.

В листинге 7.9 целевой объект Ant используется для объединения трех разных отчетов о покрытии Cobertura.

Листинг 7.9. Задача Cobertura `merge` в действии

```
    <target name="merge-coverage" depends="all-coverage-run">
        <cobertura-merge datafile="${cobertura.all.ser}">
            <fileset dir="${base.dir}">
                <include name="${cobertura.comp.ser}" />
                <include name="${cobertura.unit.ser}" />
                <include name="${cobertura.sys.ser}" />
            </fileset>
        </cobertura-merge>

        <mkdir dir="${cov.report.dir}"/>
        <cobertura-report format="html"
                datafile="${base.dir}/${cobertura.all.ser}"
                destdir="${cov.report.dir}" srcdir="${src.dir}" />
</target>
```

Покрытие и производительность

Существует один важный момент, о котором следует помнить, особенно при запуске этих процессов ночью: не выполняйте проверки в то же самое время, что и эксплуатационные испытания. Это не очень эффективно. Процесс исследования покрытия серьезно снижает производительность проверки, поэтому мы настоятельно не рекомендуем осуществлять проверки производительности, стресс-тест и проверки нагрузки одновременно.

Листинг 7.10 демонстрирует использование элемента `batchtest` задачи JUnit для выполнения набора проверок компонента совместно с проверкой покрытия. Обратите внимание, что несколько проверок (нагрузки, стресс-тест и производительности) исключены из выполняемых. Подобно стратегии разделения проверок на категории, разделение проверок покрытия по времени позволяет получать все преимущества без существенных воздействий на ваши вычислительные ресурсы.

Листинг 7.10. Элемент `batchtest` с включением проверки покрытия и исключением других проверок

```
<batchtest todir="${testreportdir}">
    <fileset dir="test/component">
        <include name="**/*Test.*" />
        <exclude name="**/*StressTest.java" />
        <exclude name="**/BatchDepXMLReportPerfTest.java" />
        <exclude name="**/BatchDepXMLReportLoadTest.java" />
    </fileset>
</batchtest>
```

Резюме

В этой главе вы изучили, как использовать мощь CI для автоматизации инспекции программного обеспечения. Независимо от того, какие аномалии есть в коде, инструмент инспекции будет регулярно оповещать о подозрительном участке, в котором наиболее вероятен риск возникновения дефекта. Инспекция и обзоры кода доказали свою эффективность в качестве механизма обнаружения дефектов; однако *непрерывное* выполнение (с использованием CI) процесса инспекции поможет уменьшить время между обнаружением рискованного участка и его исправлением.

Подобно проверке, инспекция может предоставлять разработчикам, руководству, клиентам и потенциальным клиентам количественные показатели качества. Применяя инспекции для кода, группы могут определять значения качественных параметров, чтобы гарантировать соответствие некоему пороговому значению, демонстрируя будущую эффективность программного обеспечения в среде пользователя.

Хотя *сами по себе* автоматизированные инспекторы не обнаружат всех проблем, они облегчат работу персонала и позволят группе сосредоточиться на более важных и сложных исследованиях кода. Использование автоматизированных инструментов инспекции программного обеспечения является эквивалентом *большого количества* глаз, просматривающих код. Выполнение непрерывных инспекций может реально сэкономить время группы. Инспекционные отчеты оповещают о наиболее рутинных нарушениях, оставляя интеллектуальную работу людям. Быстрые обзоры кода повышают производительность труда, а также способствуют принятию правильных решений по улучшению общего качества программной системы. В главе 9, “Непрерывная обратная связь”, продемонстрировано,

как можно использовать обратную связь этих инспекторов для облегчения коммуникации и ускорения реакции.

Табл. 7.1 подводит итог практик, описанных в этой главе.

Таблица 7.1. Практики CI, обсуждаемые в этой главе

Практика	Описание
Снижайте сложность кода	Снижайте цикломатическую сложность базового кода при помощи таких автоматизированных инспекторов, как JavaNCSS или CCMetrics, позволяющих выявлять области кода с повышенной сложностью. Организуйте запуск этих инспекторов в ходе автоматизированных построений
Осуществляйте обзоры проекта непрерывно	Задействуйте инструменты, которые помогают выявлять пакеты и сборки, жестко зависящие от других пакетов и способные снизить отказоустойчивость архитектуры
Поддерживайте организационные стандарты при проверке кода	Организуйте в ходе автоматизированного построения запуск таких инструментов, как PMD или FxCop, которые оповещают о нарушении стандартов программирования
Снижайте количество двойного кода	Снижайте объем двойного кода, запуская такие инструменты, как Simian или CPD, которые точно указывают области более частого дублирования кода на основании заданного порогового значения. Используйте данную информацию в ходе рефакторинга
Оценивайте покрытие кода	Применяйте такие инструменты, как NCover, Cobertura или Clover для выявления процента покрытия проверками строк и переходов кода. Используйте эту информацию при определении областей кода, которые подлежат более подробной проверке

Вопросы

Следующие вопросы помогут вам наладить собственный непрерывный инспекционный процесс.

- Выполняете ли вы проверку модулей спорадически, периодически или непрерывно? Как часто вы запускаете полную проверку покрытия, включая модули, компоненты и систему?
- Контролируете ли вы сложность кода?
- Непрерывно ли вы выполняете автоматизированные обзоры проекта с применением таких инструментов, как JDepend или NDepend?
- Автоматизированы ли ваши проверки кода с использованием таких средств, как PMD, Checkstyle или FxCop?
- Контролируете ли вы дублирование кода?
- Действительно ли вы способны оценить покрытие кода? Как вы реагируете на эти данные?
- Известен ли вам процент кода, который имеет достаточное количество проверок?
- Правильно ли настроено ваше построение на создание отчетов покрытия?

Глава 8

Непрерывное развертывание

Если хотите сделать дело хорошо, делайте его сами.

Английская пословица

Когда вы платите за разработку, то, естественно, ожидаете получить *работоспособное* программное обеспечение, удобное для конечного пользователя и обладающее предсказуемым поведением. Следовательно, нам, как профессионалам, имеет смысл найти безошибочный способ предоставлять конечным пользователям высококачественное, работоспособное программное обеспечение, причем в оговоренные сроки. Тем не менее мы регулярно слышим истории о "кошмаре выпуска", в ходе которого все сотрудники находились в состоянии паники, лишались сна, даже седели, и при этом далеко не всегда предоставляли конечным пользователям работоспособный новый выпуск.

Эффективное создание работоспособного программного обеспечения — собственно и есть сущность такой профессии, как разработчик. Но без успешного развертывания (установки) программное обеспечение считайте что и не существует. Сегодня программное обеспечение создается и выпускается значительно чаще, чем в прошлом, поэтому данный процесс следует отработать так же четко, как и процесс разработки. Для непрерывной интеграции необходимо непрерывное развертывание, как кульминация практик и шагов, позволяющих выпускать работоспособное программное обеспечение в любое время и в любом месте, причем, по возможности, с наименьшими усилиями.

Это вовсе не означает, что процесс развертывания так прост, как говорят одни, в то время как большинство уверено в обратном. Amazon, Google и eBay — вот только несколько примеров организаций, способных быстро выпустить работоспособное программное обеспечение. Тим О'Рейли (Tim O'Reilly) упоминает, что при разработке системы Flickr,

Web-сайта совместного использования фотографий, у него бывали дни, когда он фактически *выпускал* программное обеспечение каждые 30 минут (или около того)[1].

Как показано на рис. 8.1, развертывание программного обеспечения — это последний процесс, запускаемый кнопкой <Integrate>.

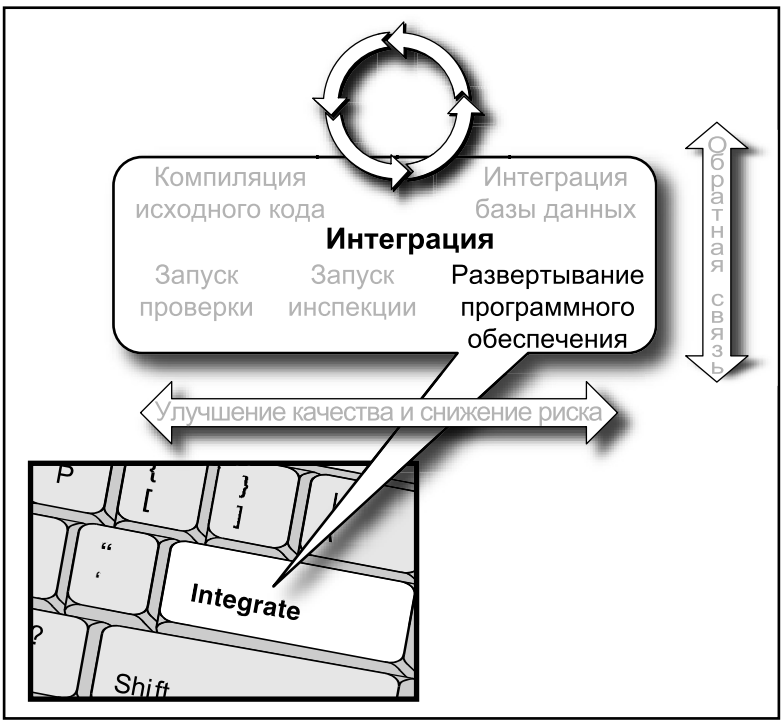


Рис. 8.1. Кнопка <Integrate> — развертывание программного обеспечения

Выпускайте работоспособное программное обеспечение в любое время в любом месте

Автоматизированное построение и воспроизводимое построение. Автоматизированные проверки и воспроизводимые проверки. Категоризация проверок и частота проверок. Непрерывные инспекции. Непрерывная интеграция базы данных. Этот набор задач по созданию эффективной среды CI предоставляет в первую очередь одно ключевое преимущество: выпуск работоспособного программного обеспечения в любой момент в любой среде. Как уже упоминалось, если вы не можете выпустить программное обеспечение, то его практически и не существует.

Из чего состоит типичное развертывание? Независимо от платформы, технологии или домена, развертывание работоспособного программного обеспечения преимущественно подразумевает шесть шагов.

[1] См. “What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software,” на `www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

1. *Маркируйте* элементы в хранилище.
2. Поддерживайте чистоту среды, *избавьтесь от предположений.*
3. Осуществляйте и *маркируйте построение* непосредственно из хранилища и устанавливайте его на выходной машине.
4. Добивайтесь успеха проверки *на всех уровнях* копии рабочей среды.
5. Создавайте отчеты *обратной связи* построения.
6. В случае необходимости *откатите* (отменить) выпуск, используя метки в хранилище с контролем версий.

Как только ваша среда CI будет установлена, некоторые зачастую болезненные этапы могут стать столь же простыми, как нажатие кнопки <Integrate>. Возможно, вам все же придется обсудить с заказчиком возможности приложения, чтобы оно полней удовлетворяло его ожиданиям, однако вы будете *знать*, что поставляемое программное обеспечение работоспособно.

Единственной вводимой командой должна быть **`ant deploy`**.

Маркируйте элементы в хранилище

Создание метки в хранилище облегчает идентификацию и отслеживание элементов, поскольку это со всей очевидностью определяет совместную принадлежность группы файлов. Но что важнее всего, маркеры позволяют отслеживать историю группы файлов, а не только отдельных файлов, версия которых может со временем меняться.

Рассмотрим, например, расположенные в хранилище файлы `Foo.cs` и `Bar.cs`, каждый для разной версии выпуска программного обеспечения (4.5 и 8.3 соответственно). Но оба выпускаются в одном пакете, если они сгруппированы в хранилище одной меткой. Как показано на рис. 8.2, маркер 3_78 указывает на общую принадлежность файлов `Foo.cs` (версия 4.5) и `Bar.cs` (версия 8.3).

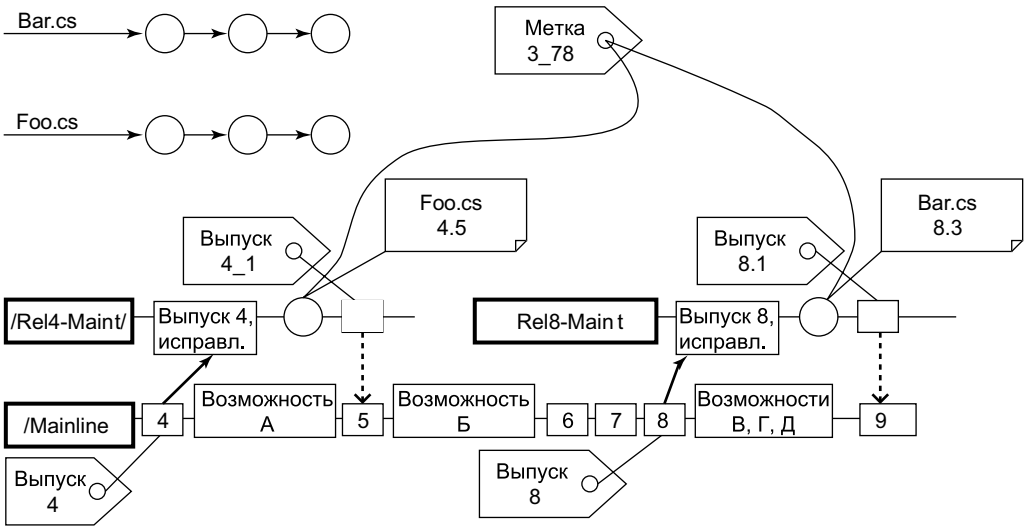


Рис. 8.2. Файлы `Foo.cs` и `Bar.cs` в том же хранилище

Маркировка версий в хранилище является первостепенным дисциплинирующим мероприятием процесса разработки программного обеспечения, гарантирующим беспроблемный переход к следующим версиям кода, а также к созданию моментальных выборок. *Моментальные выборки* (snapshot) служат основой отчетов, а в самых плохих случаях — основой отката. Применение меток позволяет также хранить внутри системы с контролем версий параллельные ветви, обеспечивая возможность поддерживать несколько линий разработки. Ниже приведен пример маркирования специфического выпуска:

```
cvs -d:pserver:uname:passwd@cvs.ib.com:/cvsrepo rtag release_9 website
```

Например, наличие параллельных ветвей с метками облегчает "исправление ошибок" выпуска. Если клиенты опробуют приложение, которое было построено под меткой `3_78`, а группа разработки работает над основной линией (самой последней версией всех файлов в системе СМ) хранилища, осуществляя магистральный выпуск с исправлением ошибок, то возникает риск, что в более новом коде появятся новые возможности или даже хуже того — новые скрытые дефекты. Но, продолжая заканчивать метку `3_78`, разработчики могут устранить обнаруженные дефекты и получить стабильное построение, которое не обязательно будет содержать новые возможности, создаваемые на основной линии.

Как только выпуск будет помечен, его создание становится очень простым. При помощи Ant, например, вы можете проверить метку группы элементов в CVS и определить идентификатор метки (см. листинг 8.1).

Листинг 8.1. Проверка метки версии в CVS

```
<cvs
cvsRoot=":pserver:${cvs.usern}:${cvs.passw}@${cvs.server.hostname}:
${cvs.server.path}" package="${cvs.module}"tag="${cvs.tag.id}"
dest="${cvs.module.dest}" command="checkout" />
```

Можно использовать разные стили маркировки версий в хранилище; но самый простой — следовать приведенному ниже шаблону именования:

`главный номер выпуска_` (или точка, если используемая система СМ допускает это) *`дополнительный номер выпуска`* (например, `2_89`)

Поддерживайте чистоту среды

Пытались ли вы когда-нибудь развернуть программное обеспечение в рабочей среде только для того, чтобы обнаружить, что она имеет неподходящую версию операционной системы, базы данных или сервера приложений? Поддержание чистой среды — это вопрос удаления и повторной установки программного обеспечения, сценариев и значений конфигурации, чтобы гарантировать ожидаемую работу среды.

При построении программного обеспечения критически важно удостовериться, что не осталось никаких устаревших или незаконченных файлов и никаких параметров конфигурации, которые могут привести к сбою программного обеспечения (или превратить отрицательный результат в положительный). Для этого используются разные подходы. Первый подразумевает полную установку системы начиная с "чистого" компьютера. Так обычно поступают на машине проверки или инсценировки. В идеале вы должны автоматизировать удаление и повторную установку каждого слоя. Например, после удаления с машины всего вам необходимо будет установить следующие слои.

- Операционная система.
- Настройка операционной системы (например, подключения к сети, настройка пользователей и брандмауэра).
- Серверные компоненты программного обеспечения (например, сервер приложений, сервер баз данных и сервер обмена сообщениями).
- Настройка сервера.
- Инструменты стороннего производителя (типа Web-среды, инструментов объектно-реляционного связывания и т.д.).
- Специальное программное обеспечение (программное обеспечение, написанное для заказчика).

Но при построении программного обеспечения можно удалять только один слой, например компонентов разрабатываемого программного обеспечения. Количество удаляемых и повторно устанавливаемых слоев зависит от допустимого уровня риска. Если приложение полагается на несколько разных файлов операционной системы, то, возможно, имеет смысл чаще переустанавливать всю систему. В любом случае мы рекомендуем перезагрузку всех слоев хотя бы несколько раз, по крайней мере перед выпуском программного обеспечения конечным пользователям.

Маркируйте каждое построение

Создание уникального идентификатора для построения в виде метки построения подразумевает два этапа: сначала маркировка кода в хранилище (как было описано только что), а затем фактическое построение с использованием уникальной метки. Метки построения не только дают общее представление о версии кода определенной системы, а и помечают дефекты, возможности и новые требования, которые могут быть предъявлены к данному экземпляру базового кода.

Обратите внимание на различие между меткой хранилища и меткой построения. Метки хранилища связывают в группу файлы (обычно неоткомпилированные). Метки построения уникально идентифицируют бинарный результат построения. Это может быть набор исполняемых файлов, файл `.jar`, сборка .NET или даже файл `.zip`. Обычно для меток построения используют две схемы именования, учитывающие номера сборки и платформы. Убедитесь, что все участники рабочего проекта знают и следуют соглашению об именах меток. Например, если код в хранилище имеет метку `2_89`, то построение этого кода могло бы иметь метку `2_89.01`. Если построение планируется под определенную платформу, то метка может содержать дополнительную информацию, например `2_89.hp-01`.

Отсутствие меток построения затрудняет ассоциацию с возможностями, дефектами и требованиями к бинарному артефакту. Например, развертывание непомеченного построения в среде проверки, такой как QA, в сущности создает перемещаемый объект. Если группа QA обнаружит дефект, то координация с группой разработки может оказаться проблематичной. Без возможности точно указать, где и когда возникла проблема, становится трудно идентифицировать то, что к ней привело. Но если построение помечено, сообщение о проблеме становится вопросом указания уникального идентификатора построения.

Маркировка построения столь же проста, как и выполнение полного построения с назначением ему идентификатора. Например, для развертывания помеченной версии в среде QA достаточно ввести следующую команду:

```
ant -Dbuild.id=2_89.01 -Denvironment=qa deploy
```

В автоматизации развертывания интересней всего то, что все другие процессы (компиляция, интеграция базы данных, проверка, инспекция и т.д.) должны быть выполнены *перед* ним. Компиляция, безусловно, перед развертыванием выполняется, но существуют и другие ключевые этапы процесса, включая перестроение базы данных, а также успешные проверки и инспекции. Кроме того, после развертывания не мешало бы запустить дополнительные проверки.

Запускайте все проверки

В то время как некоторые текущие этапы разработки могут требовать выполнения только некоторых групп проверок, перед построением, предназначенным для упаковки развертывания, следует *запустить все проверки, которые должны пройти успешно*. Это так просто. Выполните все автоматизированные проверки, от проверок модулей до проверок функций. Это можно сделать за один раз на машине построения, однако важнейшей составляющей проверки перед развертыванием является запуск всех проверок в чистой, заново установленной среде, представляющей собой клон рабочей среды. Важно еще раз удостовериться, что никакие проблемы с окружением не приведут к отказу или неожиданному снижению производительности. Запуская все проверки перед переходом к следующему рабочему этапу, вы повышаете свою уверенность в том, что получите *работоспособное программное обеспечение*. И даже при том, что мы уверены в мощности и необходимости автоматизации всех видов процессов, включая проверку, программное обеспечение остается продуктом, который будет использоваться людьми, а следовательно, он все еще нуждается в проверке именно ими.

Создавайте отчеты обратной связи построения

Организация обратной связи автоматизированного построения облегчает общее понимание того, *что именно* происходит в его процессе, включая различия в файлах построения, обнаруженные дефекты и реализацию возможностей. Фиксируя эту информацию, заинтересованные стороны могут проверить наличие или отсутствие оговоренных аспектов.

Не отказывайтесь от человеческого фактора

Вы могли бы возразить, что даже наиболее надежная автоматизированная проверка все же остается “вещью в себе”. Чтобы убедиться в том, что ваш продукт ведет себя так, как ожидает заказчик, перед выпуском программного обеспечения вы должны рассмотреть его “извне”, как бы подражая действиям пользователя.

Однажды я разговаривал с руководителем проектов большого финансового учреждения, чья группа разрабатывала довольно строгую автоматизированную систему контроля поведения. Они обеспечили высокую степень проверки на всех уровнях и создали довольно надежный процесс автоматического развертывания своих построений. Однако они обратили внимание на то, что при некоторых развертываниях их приложения в рабочей среде компании пользовательский интерфейс явно имел определенные проблемы, такие как страницы с нарушенными таблицами и отсутствием изображений. Это было особенно болезненно для руководителя, поскольку ему пришлось выяснять отношения по поводу этих проблем с *другими* группами компании, которые зависели от данного приложения. Оказалось, что группа разработки так сосредоточилась на автоматизации, что несколько увлеклась: фактически никто и никогда не проверял поведение и внешний вид продукта лично. Мы ни в коем случае не осуждаем автоматизированную проверку, однако просмотр приложения вручную позволяет выявить такие вещи, которые “робот”

пропустит. Если бы они ввели проверки вручную, то проблемы с UI в значительной степени были бы сняты.

Проверка человеком требует успеха на 100%. Ее провал может быть вызван незначительными проблемами в среде или базовом коде, который впоследствии после развертывания приложения мог бы привести к катастрофе.

98% успеха, это годится?

Однажды я консультировал организацию, в которой порог успеха проверок составлял 98%. Эта стратегия была принята потому, что в связи с различными сложностями базового кода и среды организации никогда не удавалось достичь 100% успеха в любой момент времени. К сожалению, подобная стратегия, когда допускается не 100% успеха, создала ситуацию неопределенности между построениями — разработчики не имели никакого представления, какие именно проверки потерпели неудачу между выпусками и были ли это те же самые отказы или новые. Подход применения CI требует *автоматизации* проверок и их стопроцентного успеха, чтобы предоставить разработчикам данные о том, какие проверки потерпели неудачу и почему.

Например, когда последняя бета-версия программы передается группе QA, которая должна выяснить, на самом ли деле были устранены первостепенные дефекты, наличие отчета, где явно указаны обнаруженные ранее дефекты, позволит ускорить и облегчить этот процесс.

Еще один отчет, который работает в тандеме с отчетом об обнаруженных дефектах, — это отчет о различиях в файлах, создаваемый при построении. Данный отчет облегчает понимание того, что изменилось в системе с контролем версий при построении. Листинг 8.2 демонстрирует пример, как на основании двух дат или меток инструмент построения наподобие Ant может создать при построении список различий между файлами в хранилище CVS и поместить его в отчет.

Листинг 8.2. Создание отчета о различиях при помощи Ant

```
<target name="diff-tag-to-tag">
    <delete dir="${cvs.reports.dir}" />
    <mkdir dir="${cvs.reports.dir}" />
    <cvstagdiff package="${cvstagdiff.package}"
            destfile="${cvstagdiff.destfile}"
            starttag="${cvstagdiff.starttag}"
            endtag="${cvstagdiff.endtag}"compression="true" />
    <style in="${cvstagdiff.destfile}"
            out="${cvs.reports.dir}/tagdiff.html"
            style="${ant.home}/etc/tagdiff.xsl">
        <param name="title" expression="Ant Diff" />
        <param name="module" expression="ant" />
        <param name="cvsweb"
            expression="http://cvs.ib.com/viewcvs/"/>
    </style>
</target>
```

Задача Ant создаст отчет HTML, который описывает версии всех измененных файлов. Используя этот отчет, заинтересованные люди могут быстро отслеживать изменения по версиям. Представьте, например, что группа QA получила отчет о ряде дефектов в предыдущей бета-версии программы. При последующем выпуске группа QA сообщает, что

обнаружен дефект, идентичный тому, который, как считалось, был "устранен" в предыдущем построении. Сотрудничая с группой разработки, группа QA может исследовать отчет о различиях построения, выявить связанные с ним изменения и устранить этот дефект фактически.

Позаботьтесь о возможности отката выпуска

В конечном счете, наличие возможности *отменить* (undo) развертывание — важнейшая часть эффективной разработки. Многие группы сталкивались с ситуацией, когда для быстрого устранения вновь замеченного дефекта им приходилось заменять код предыдущим выпуском, который работал лучше. При использовании маркировки построения и хранилища этот процесс требует лишь указания желаемой версии.

Скажем, например, что в течение планового выпуска группа QA получает построение `89_3.04`, в котором должен быть устранен ряд первоочередных дефектов. Однако после развертывания группа QA быстро выясняет, что эта последняя бета-версия программы имеет небольшой дефект, который, тем не менее, не позволяет провести дальнейшую проверку. Оперативно вернувшись к предыдущему построению (`89_3.03`), группа QA не обязательно потеряет драгоценное время проверки и сможет продолжить проверку предыдущего выпуска и искать в нем дефекты.

Резюме

Ваш проект может извлечь пользу из непрерывного развертывания работоспособного программного обеспечения. В то время как к каждому приложению, платформе и домену предъявляются индивидуальные требования, процесс эффективного выпуска работоспособного программного обеспечения в любое время и в любом месте в значительной степени зависит от шести этапов. Маркировка версий в хранилище объединяет группу взаимосвязанных файлов, создание чистой среды уменьшает количество предположений о существующих элементах, способных затруднить даже самое простое из построений. В результате маркированного построения получается именованный бинарный код, отчет о котором позволяет узнать, что все проверки выполнены успешно. Это дает вам больше уверенности в том, что программное обеспечение будет работать как ожидалось. Отчеты обратной связи построения облегчают группе общее понимание возможностей, дефектов и требований, предъявляемых к бинарному коду. Наконец, наличие возможности отката выпуска означает, что если что-нибудь пойдет уж совсем не так, как надо, вы можете снабдить пользователей последней рабочей версией.

Табл. 8.1 подводит итог практик, описанных в этой главе.

Таблица 8.1. Практики CI, обсуждаемые в этой главе

Практика	Описание
Выпускайте работоспособное программное обеспечение в любое время в любом месте	Выполнение полностью автоматизированного построения, включающего компиляцию, все проверки, инспекции, упаковку и развертывание, позволяет выпускать работоспособное программное обеспечение в любое время и в любой известной среде
Маркируйте элементы в хранилище	Маркируйте файлы проекта в хранилище с контролем версий. Как правило, это делается в конце промежуточного этапа проекта

Окончание табл. 8.1

Практика	Описание
Поддерживайте чистоту среды	Удалите с машины интеграционного построения файлы, изменения конфигурации, серверы и все остальное, чтобы обеспечить интеграционному построению условия для успеха. Чем строже этот процесс, тем лучше
Маркируйте каждое построение	Маркируйте бинарные артефакты построения и развертывания в хранилище с контролем версий
Запускайте все проверки	Выполните все проверки программного обеспечения. Сюда относятся проверки модуля, компонента, системы, функций, а, возможно, даже производительности, нагрузки и другие типы проверок, которые гарантируют, что программное обеспечение готово к переходу на следующий этап рабочего процесса или даже поставке заказчику
Создавайте отчеты обратной связи построения	Документируйте изменения, которые были сделаны в последнем построении. Это может оказаться полезным для других групп, например QA
Позаботьтесь о возможности отката выпуска	Что-нибудь всегда может пойти не так, как надо, поэтому используйте метки построения для возможности отмены любых изменений, которые не следовало передавать в хранилище с контролем версий

Вопросы

Автоматизировано ли ваше развертывание? Как быстро вы способны получить рабочий выпуск? Как быстро вы сможете установить программное обеспечение в среду разработки или проверки? Используйте эти вопросы, чтобы определить возможность вашего проекта непрерывно развертывать программное обеспечение.

- Обладаете ли вы возможностью к откату выпуска?
- Маркируете ли вы свои построения в системе с контролем версий?
- Имеете ли вы полный набор автоматизированных проверок, за которыми следуют проверки последней бета-версии программы вручную?
- Как ваша группа справляется с исправлением выпуска?
- Связаны ли ваши метки версий и построений?
- Создается ли при построении отчет обратной связи?
- Действительно ли вы способны развернуть ваше программное обеспечение из командной строки одной командой?
- Осуществляется ли построение и развертывание из хранилища с контролем версий?
- Действительно ли вы способны настроить развертывание для различных сред?
- Устанавливается ли ваше программное обеспечение и запускается на “чистом” клоне среды пользователя?
- Имеете ли вы систему отслеживания ошибок, способную создавать отчеты?

Глава 9

Непрерывная обратная связь

Используйте механизмы непрерывной обратной связи

Как правило, наиболее успешный человек в жизни — это человек, обладающий наилучшей информацией.[1]

Бенджамин Дисраели (Benjamin Disraeli) (1804–1881)

Однажды во время разговора с моим коллегой Чаком поступила информация о неудаче построения одного из наших проектов. Я узнал об этом, поскольку получил SMS на свой мобильный телефон, на моем компьютере сыграл соответствующий звуковой файл, пришло сообщение по электронной почте, и шар на моем рабочем столе изменил цвет на красный. Я прервал диалог с Чаком и позвонил техническому руководителю проекта, от которого услышал: "Я только что получил такое же сообщение и как раз занимаюсь этим". Я положил трубку и продолжил диалог не отвлекаясь. Это и есть пример непрерывной обратной связи в действии. Я был вынужден прервать Чака лишь на мгновение, причем это иллюстрация относительно скромных типов механизмов обратной связи, которые вы можете использовать в вашей системе CI.

Обратная связь — это ключевой результат нажатия кнопки <Integrate>. Без обратной связи все остальные аспекты CI бесполезны. Причина, по которой построение должно проходить быстро и так же быстро терпеть неудачу — это и есть обратная связь. Быстрая обратная связь — основа CI. Например, если вы не можете узнать о неудаче проверки или инспекции на протяжении нескольких часов, то вы не сможете и устранить проблему немедленно, прежде чем она приведет к другим сбоям. То же относится и к отказу базы данных или неудаче развертывания. Обратная связь нужна для принятия мер, она предоставляет информацию о реальным текущем состоянии вашего интеграционного построения (рис. 9.1).

Информация о проекте программного обеспечения постоянно изменяется. Как заметил древнегреческий философ Гераклит: "Все течет, все изменяется"[2]. Мы должны обмениваться информацией с нашими клиентами, разработчиками, начальством и всеми остальными участниками проекта, причем жизненно необходимо, чтобы это проходило быстро и

[1] Владеющий информацией владеет миром (мне автор неизвестен). — *Примеч. ред.*

[2] См. `http://www.aphorism.ru/author/a4213.shtml`. — *Примеч. ред.*

кратко. Хотя личное общение, наверное, наиболее эффективно, оно *не очень* масштабно. Организация *непрерывной* обратной связи обеспечивает группу возможностью автоматически оповещать большое количество людей о состоянии проекта; кроме того, такая информация может быть объединена и исследована на предмет выявления тенденций проекта.

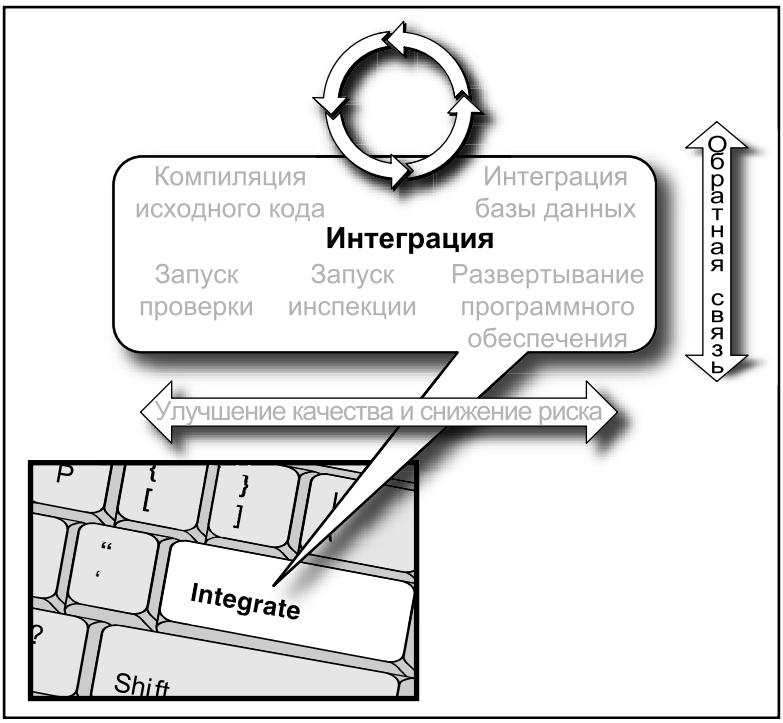


Рис. 9.1. Поддержка обратной связи кнопкой <Integrate>

В этой главе рассматривается, как и кому посылать необходимую информацию.

Вся необходимая информация

Если посылать информацию каждый раз и всем подряд, то это не даст результатов. Задача обратной связи заключается в том, чтобы как можно быстрее и точнее уведомить о событии. Необходимо передать *правильную информацию правильным людям* в *правильное время* и *правильным способом*. Сначала мы рассмотрим, что подразумевается под правильной информацией, каков ее тип и как гарантировать ее точность. Затем обсудим, кто должен получать информацию, почему и когда.

Непрерывная обратная связь и CI

Предоставление правильной информации правильным людям в правильное время и правильным способом — для решения этой задачи CI — наилучший инструмент, поскольку он позволяет наладить персонализированную автоматизированную обратную связь в масштабе реального времени (непрерывно).

И наконец, при обсуждении правильного способа передачи информации мы рассмотрим некоторые из механизмов связи, которые могут быть использованы системой CI и известны как *устройства непрерывной обратной связи* (Continuous Feedback Device — CFD). Обычно устройства CFD сообщают участникам проекта об успехе или отказе построения, но они могут также оповещать и о других проблемах. Например, устройство CFD может сообщать заинтересованным сторонам о превышении некоторого порогового значения (например, дублирования кода).

Данный подход иллюстрирует рис. 9.2.

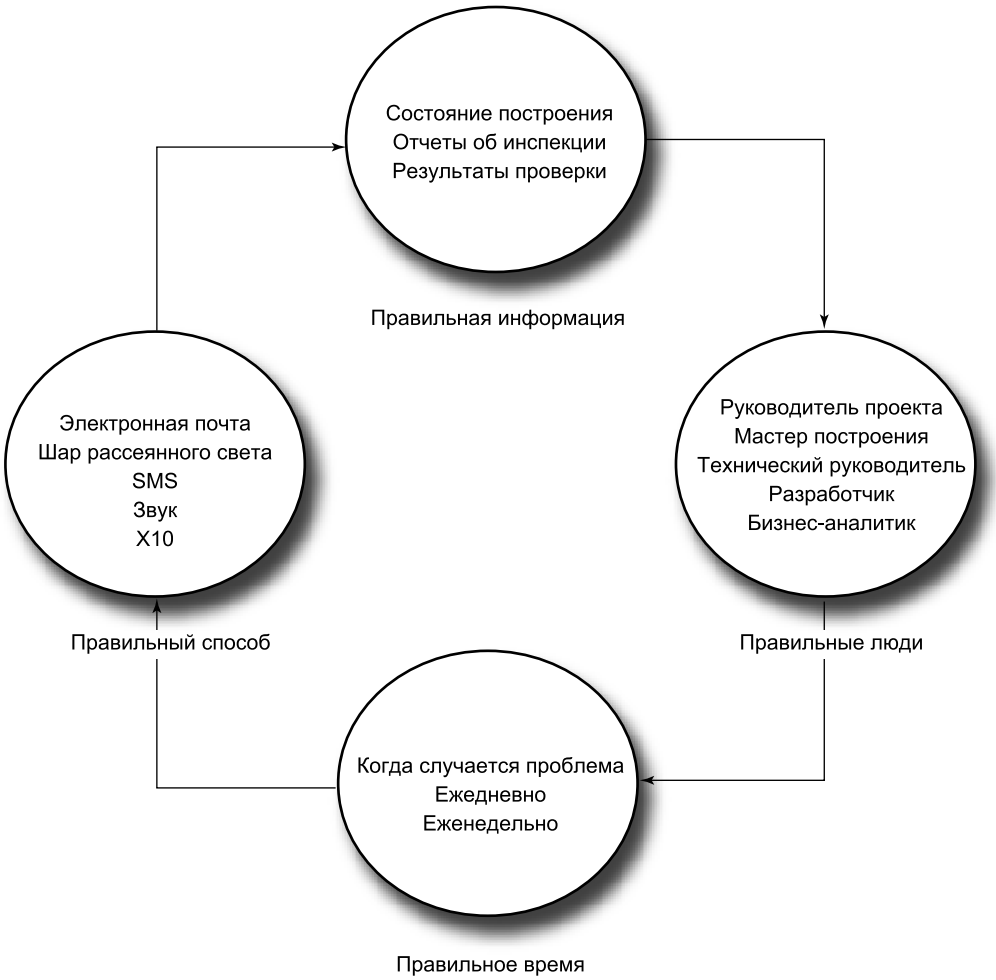


Рис. 9.2. Правильные материалы непрерывной обратной связи

Правильная информация

Непрерывная обратная связь сама по себе *не предпринимает* никаких действий по улучшению программного обеспечения, это делают участники проекта, обычно разработчики ПО. Информация может быть полезна и другим людям, например аналитикам, ис-

пытателям и руководству. Непрерывная обратная связь предоставляет средства для структурирования информации, а вы уже решаете, какие ее части, каким участникам проекта и при каких обстоятельствах передавать. Уверенность в том, что данная информация точна и нова, позволяет вам принять наиболее эффективные меры по решению проблемы. Чтобы не ждать, когда у участников проекта возникнут вопросы, которые затем неизбежно будут переадресованы вам, *вы* можете создать для них уведомления по специфическим случаям и рассылать их регулярно или при возникновении проблем.

Как уже неоднократно упоминалось в этой книге, систему CI можно задействовать для выполнения тех работ, которые в противном случае вам пришлось бы делать вручную. Уведомление о состоянии построения — это хороший пример информации, актуальной для большинства участников проекта. В идеале такое сообщение включает результаты всех регрессионных проверок, примененных к приложению, всех инспекций, оповещающих об аномалиях в исходном коде, и результаты развертывания. Не всю эту информацию следует рассылать всегда, можно использовать некоторый тип расписания.

Правильные люди

Все участники проекта должны получать обратную связь определенного типа, но не обязательно включающую все ее элементы *каждый раз*. Немного занявшись планированием, вы можете выработать стратегию распределения информации по типам. Иногда сообщения просто уведомляют участников об успехе, однако не стоит рассчитывать на них всегда. Частая отправка сообщений обратной связи всем участникам проекта может вызвать у них чувство неприятия. Кроме того, уведомление всей группы о том, что в силах исправить один или два ее участника, создает лишь ненужный информационный трафик. Зачем получать сообщение об ошибке всем разработчикам, если ею может заняться один конкретный человек? Среди большого количества сообщений не мудрено пропустить относящиеся непосредственно к вашей области. Важно, чтобы группа не стала игнорировать уведомления CI о процессе построения в целом.

Остерегайтесь перегрузки информацией

Посылка сообщений обратной связи всем участникам проекта, как правило, приводит к тому, что их начинают игнорировать.

Система CI помогает предоставить достоверную информацию именно тем людям, которым она действительно необходима. В некоторых проектах один человек может выполнять несколько задач. В зависимости от вашей роли в проекте вы можете получить сообщение от системы CI разными способами.

- **Руководитель проекта**. *Руководитель проекта* (Project Manager — PM) зачастую должен принимать решения на основании состояния ресурсов (людских, аппаратных средств и содержимого хранилища), времени и цен. PM обычно решает множество задач сразу, поэтому он нуждается в обратной связи высокого уровня в масштабе реального времени, поскольку это непосредственно относится ко времени, цене, качеству и контексту. В связи с автоматизированным и непрерывным характером работы система CI может оказаться весьма эффективной для обеспечения такой обратной связи.
- **Архитектор и технический руководитель**. Технические руководители и архитекторы обычно хотят видеть общее состояние всего построения, поскольку они рассматривают всю систему. Специфический интерес для них представляют качественные

показатели, получаемые в результате работы инструментальных средств статического и динамического анализа, свидетельствующие о соблюдении стандартов в коде и архитектуре.

- **Разработчики**. Как правило, разработчики получают от системы CI сообщения о коде, только что проверенном в хранилище с контролем версий. Разработчики получают разнообразную информацию о результатах проверок и инспекций, а также о состоянии последних построений. Группа извлекает пользу из сообщений, получаемых каждым от механизма непрерывной обратной связи (например, по электронной почте), только тогда, когда они относятся к их собственных задачам.
- **Испытатели**. Испытатели, по всей вероятности, больше всех заинтересованы в результатах автоматизированных проверок и инспекций, передаваемых по обратной связи. Сообщения для них содержат информацию обо всех проверках и инспекциях кода системы. В зависимости от подхода проверки, принятого в вашей группе, эта информация может использоваться для изучения новых возможностей, прежде чем они будут переданы группе проверки.

Правильное время

Устаревшие новости — это уже не новости. Информация о том, что построение потерпело неудачу два дня назад, не особенно полезна. Но вот ее немедленная передача — хороший повод настройки непрерывной обратной связи на передачу информации в *правильное время*. Как уже установили эксперты, уменьшение периода между проявлением, обнаружением причины и устранением дефекта экономит время и деньги. Чем больше времени проходит между проявлением дефекта и оповещением о нем, чем меньше ответственных сторон узнает о случившемся, тем выше вероятность применения дефектного принципа в другом месте. Вокруг него могут построить другой компонент или "устранить" дефектную часть исходного кода. Безусловно, потраченное впустую время и деньги важны, но еще важней возможность устранения ошибок, которых, в принципе, может и не быть.

Основа непрерывной обратной связи

Основой непрерывной обратной связи является уменьшение времени между проявлением дефекта, обнаружением его причины и устранением.

CI чрезвычайно эффективна для предоставления правильной информации правильным людям в правильное время. Используя сервер CI, такой, например, как CruiseControl, можно обеспечить немедленное распространение информации при сбое или успехе построения наряду со всем доступным богатством ее детализации, которая способствует решению этой и других проблем.

Правильный способ

Система CI предоставляет также возможность передавать информацию *правильным способом*, подразумевающим наиболее подходящий механизм связи для ее представления. Существуют различные механизмы непрерывной обратной связи, например электронная почта, устройства звукового и визуального оповещения, а также текстовые сообщения.

Некоторые механизмы обратной связи предпочтительнее. Иногда годится лист бумаги на стене, содержащий перечень текущих дефектов, — это весьма эффективное средство коммуникации. Но это также яркий пример средства, информация которого может очень

быстро устареть, поэтому в данной главе мы в основном сосредоточимся на использовании автоматических способов оповещения в реальном масштабе времени. Обратная связь обычно предписывает некий тип действий, поэтому ее сообщения должны поступать правильным способом и нужным людям, которые могут иметь различные предпочтения или потребность в обратной связи.

Используйте механизмы непрерывной обратной связи

Подобно тому как вам понадобился бы молоток не для всякого мелкого ремонта по дому, вы не всегда будете использовать тот же механизм непрерывной обратной связи для всех случаев. Этот раздел знакомит вас с рядом механизмов, способных обеспечить непрерывную обратную связь: электронная почта, обмен текстовыми сообщениями, шар рассеянного света (устройство стандарта X10), панель задач Windows на мониторе, звук и т.д. Обсуждается также использование широкоэкранных мониторов.

Рис. 9.3 демонстрирует различные механизмы обратной связи, которые вы можете применять в системе CI.

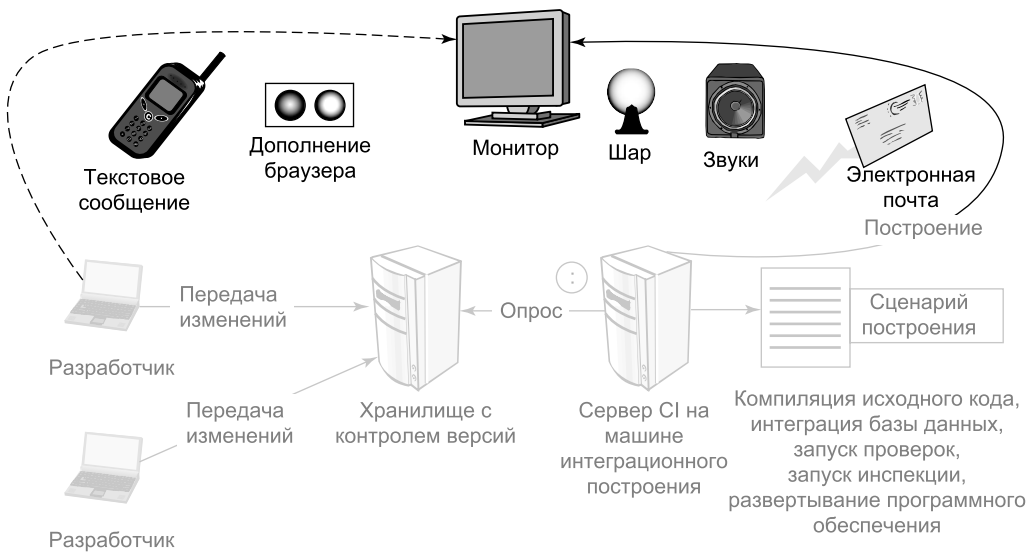


Рис. 9.3. Механизмы непрерывной обратной связи

Электронная почта

Рассматривая электронную почту как механизм обратной связи, вы должны учитывать следующие требования, преимущества и недостатки.

Требования: сервер CI типа CruiseControl, клиент электронной почты типа Microsoft Outlook или Eudora и сервер электронной почты (поддерживающий протокол SMTP) типа James[3].

[3] Почтовый сервер Apache Java Enterprise Mail Server (он же Apache James) — это написанный полностью на Java сервер SMTP, POP3 и NNTP. См. `http://james.apache.org/`.

Преимущества: асинхронно передает информацию необходимым людям.

Недостатки: многие не всегда регулярно проверяют электронную почту, а также создается потенциальная возможность спама.

Электронная почта — это наиболее популярная форма обратной связи систем CI. Последние в случае успеха или неудачи построения могут посылать сообщения электронной почты, содержащие заранее определенные подробности. Я, например, настраиваю сервер CruiseControl так, чтобы отправлять по электронной почте сообщения в формате HTML, отображающие состояние построения, изменения с момента последнего построения, результаты проверок модуля, а также информацию о файлах, созданных в ходе развертывания. Сюда также включается ссылка на приложение отчетов сервера CruiseControl, где я могу просмотреть подробную информацию о построении, включая доступ к инспекционным артефактам и графикам тенденций. Электронная почта — весьма полезная форма обратной связи, но она имеет и недостатки. Нет эффективного способа поддерживать актуальность информации о тенденциях программного обеспечении без риска наводнить сеть сообщениями электронной почты, которые рассылаются при каждом изменении.

Листинг 9.1 демонстрирует файл `config.xml` сервера CruiseControl, настраивающий его на отправку электронной почты человеку (`@localhost`), последним передавшему измененные файлы, и техническому руководителю проекта (`pduvall@localhost`) при помощи атрибута `default-suffix` элемента `htmlemail`.

Листинг 9.1. Файл `config.xml` сервера CruiseControl, настраивающий передачу электронной почты

```
...
<publishers>
    <currentbuildstatuspublisher
        file="buildstatus.txt"/>
    <htmlemail mailhost="localhost"
        xslDir="xsl"
        css="cruisecontrol.css"
        returnaddress="buildstatus@localhost"
        returnname="ABC Project Build Status"
        defaultsuffix="@localhost"
        spamwhilebroken="true"
        buildresultsurl="http://localhost:8989/cruisecontrol>
        <always address="pduvall@localhost"/>
        <failure address="pduvall@localhost "/>
    </htmlemail>
</publishers>
```

Рис. 9.4 демонстрирует пример сообщения о состоянии построения в формате HTML, которое вы можете получить от сервера CI. Обратите также внимание на то, что в папке Входящие (Inbox) содержатся три сообщения, полученные за пару минут. Большинство людей (включая вас) могут устать от таких сообщений и начать игнорировать подобную форму обратной связи.

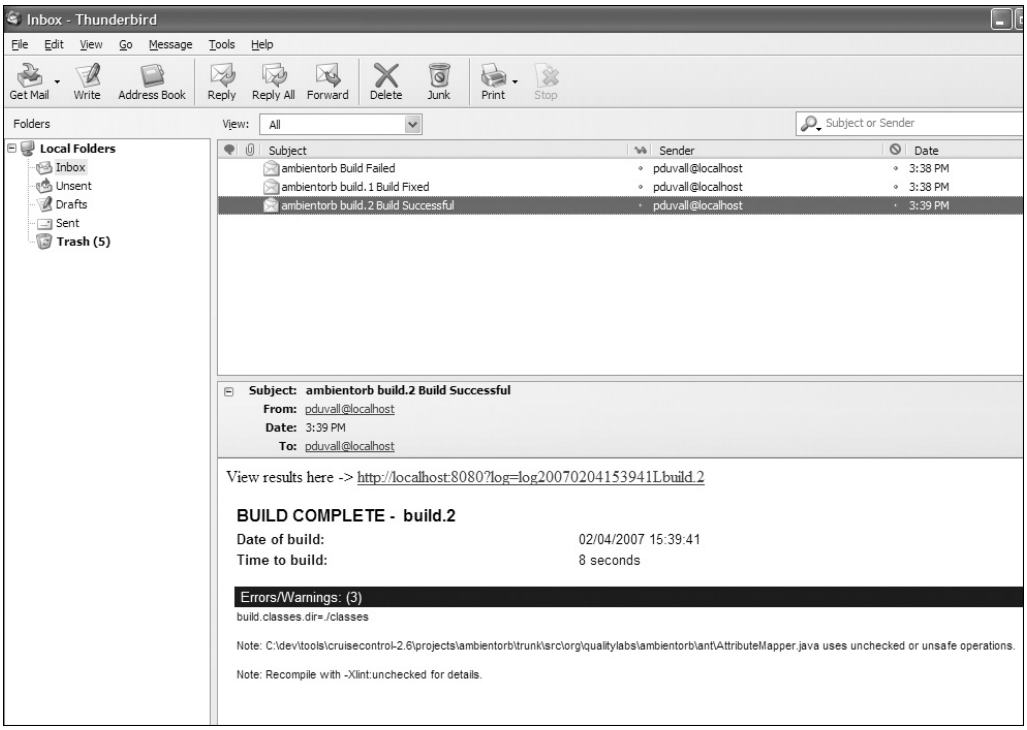


Рис. 9.4. Сообщение электронной почты о состоянии построения

SMS (текстовые сообщения)

Рассматривая SMS в качестве механизма обратной связи, учитывайте следующее.

Требования: мобильный телефон с функцией SMS, сервер электронной почты и инструмент, способный ее передавать.

Преимущества: возможность получать сообщения, находясь далеко от средств электронной почты.

Недостатки: краткость сообщений. Те же недостатки, которые упоминались для электронной почты.

Имея сервер CI, довольно легко заставить сервер электронной почты отправлять SMS на мобильный телефон. Все, что необходимо — это телефон с функцией SMS и инструмент, способный передавать сообщения через сервер электронной почты. На сервере CruiseControl это так же просто, как передать сообщение электронной почты. Листинг 9.2 демонстрирует передачу текстового сообщения сервером CruiseControl в случае сбоя.

Листинг 9.2. Передача текстового сообщения сервером CruiseControl при сбое построения

```
<publishers>
    <email mailhost="smtp.mydomain.com"
        returnaddress=buildstatus@mydomain.com
        defaultsuffix=@mydomain.com
        returnname="Project Build Status"
```

```
        spamwhilebroken="false"
        buildresultsurl="SMS">
        <failure address="7035551212@mobilephone-emailaddress.com"
            reportWhenFixed="true"/>
    </email>
</publishers>
```

Вот короткое объяснение кода листинга 9.2 и некоторых других параметров, которые вы можете использовать при передаче SMS.

- Строку `smtp.mydomain.com` следует заменить адресом вашего сервера SMTP. Для передачи электронной почты через сервер SMTP вам может понадобиться указать идентификационную информацию; в данном случае можно использовать атрибуты `username` и `password`.
- Атрибут `returnaddress` задает обратный адрес электронной почты, который в сообщении указывается в поле `from`.
- Текст *`@mydomain.com`* в атрибуте `defaultsuffix` следует заменить на имя вашего домена.
- Дочерний элемент `<failure>` в данном случае используется для передачи сообщения об отказе по определенному адресу электронной почты. Значение *`7035551212@mobilephone-emailaddress.com`* следует заменить на адрес электронной почты человека, которого необходимо уведомить об отказе построения. Установка для атрибута `reportWhenFixed` значения `true` указывает на необходимость послать по электронной почте дополнительное сообщение после исправления построения.
- Сервер CruiseControl позволяет применять атрибут `xslfile`, который устраняет необходимость в атрибутах `css` и `xslDir`.

Вы можете изъявить желание получать текстовые сообщения о каждом успешном или сбойном построении. Однако я предпочитаю получать уведомление только о сбое построения, а затем еще одно о его исправлении. У нас происходит по нескольку построений в день, и мне вовсе не нужен поток текстовых сообщений, на которые не нужно реагировать (т.е. построения проходят успешно).

Шар рассеянного света и устройства стандарта X10

Визуальные устройства прекрасно подходят для уведомления, поскольку их можно установить где угодно (некоторые даже не нужно подключать к компьютеру) и участникам группы будет достаточно просто посмотреть на устройство, чтобы определить состояние построения.

Шар рассеянного света

Мы рекомендуем использовать *шар рассеянного света* (Ambient Orb), поскольку он может быть настроен на отображение большого количества различных цветов, позволяющих уведомлять вас о разных событиях. Это не самое дешевое устройство X10, так что ваша группа сперва должна оценить его возможности. Рассматривая шар рассеянного света в качестве механизма обратной связи, учитывайте следующее.

Требования: шар рассеянного света — это специальное предложение от Ambient Devices[4], которому необходим сценарий, способный передавать сообщения HTTP `get`, сце-

[4] См. *`www.ambientdevices.com`*.

нарий построения (например, Ant) и сетевое соединение (или 9-штырьковый последовательный разъем).

Преимущества: "мгновенная", нецифровая информация; замечательная игрушка.

Недостатки: цена, информация предоставляется без подробностей; вы должны находиться в зоне прямой видимости, чтобы обратить внимание на его действие.

Мы называем шар "визуальным устройством" потому, что вам достаточно просто посмотреть на него, чтобы определить состояние проекта. Вовсе не обязательно получать 20 сообщений электронной почты, содержащих информацию о различных пороговых значениях, которые проект, возможно, превысил, и без остальных подробностей. Каждый участник проекта может посмотреть на шар и узнать состояние последнего построения, а также его качественные показатели (если вы добавили соответствующие настройки). Я установил шар в нашей комнате разработки проекта так, чтобы он отображал различные цвета в зависимости от состояния построения. Например, если построение потерпело неудачу и не исправлено в течение 30 минут, то он становится ярко-красным.

Преимущество использования шара заключается в том, что он может быть помещен в любом месте и будет предоставлять не цифровую информацию, как электронная почта или устройства X10 (обсуждаемые далее). Шар содержит приспособление наподобие пейджера, работающего по беспроводной сети. Web-серверному компоненту устройства из сценария построения, например Ant, может быть послано сообщение HTTP `get`. Возможно, шары несколько дороговаты, но они доказали свою безусловную полезность в проектах. Это, несомненно, "забавный фактор", и мы полагаем, что это демонстрирует наше весьма серьезное отношение к качеству, поскольку мы хотим немедленного уведомления о проблеме, а также вносим некий забавный элемент в рабочую обстановку.

Листинг 9.3 демонстрирует целевой объект Ant, который выполняет задачу `ambientorb`[5] и изменяет цвет шара в зависимости от успеха или отказа построения. Шар рассеянного света представлен на рис. 9.5.

Листинг 9.3. Регистрация задачи Ant `ambientorb`, позволяющей уведомлять шар рассеянного света

```
<target name="registerOrb" if="is.integration.machine">
<taskdef classname="org.qualitylabs.ambientorb.ant.OrbTask"
        name="orb" classpathref="orb.class.path"/>
    <orb listener="org.qualitylabs.ambientorb.ant.OrbListener"
        deviceId="AAA-99A-AAA"
        colorPass="green"
        colorFail="red"
        animationFail="heartbeat"
        animationPass="none"
        commentPass="The+build+passed"
        commentFail="Build+Failure!!"/>
</target>
```

[5] Задача Ant для шара рассеянного света, доступен по адресу *www.qualitylabs.org/projects/ambientorb*

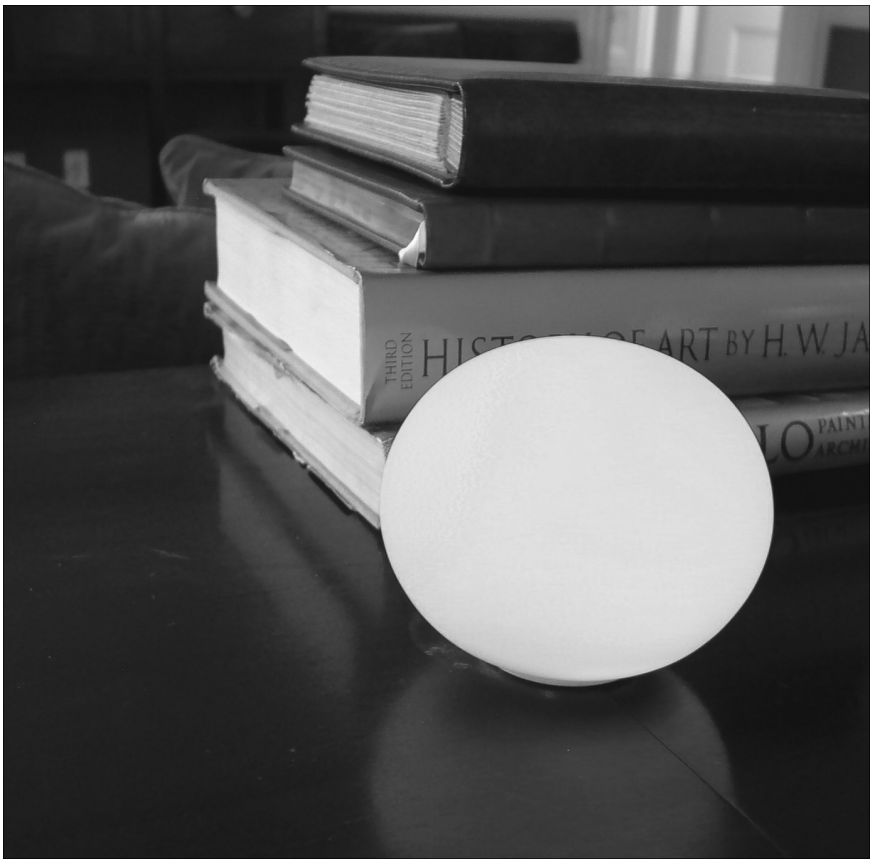


Рис. 9.5. Шар рассеянного света на столе

Устройства X10

Устройства стандарта X10 не столь гибки, как шар рассеянного света, но они составляют ему прекрасную альтернативу в случае ограниченности бюджета. Рассматривая возможность реализации устройств X10 в качестве механизма обратной связи, учитывайте следующее.

Требования: устройство, способное получать сообщения X10 и комплект автоматизации дома (например, FireCracker).

Преимущества: замечательная "визуальная" игрушка и доступ к любому электротехническому устройству.

Недостатки: информация бинарная — устройство либо включено, либо выключено. Подобно шару, чтобы обратить внимание на действие устройства, вы должны находиться в зоне прямой видимости. Устройства X10 не очень полезны для визуального оповещения.

Устройства X10 — это еще одно "визуальное" приспособление с двумя основными отличиями. На выбор доступны буквально сотни различных устройств X10, но они имеют только два визуальных режима: либо включено, либо выключено (вы сами настраиваете, что означает каждое состояние). Результат подобен шару рассеянного света: по его состоянию

можно судить о происходящих событиях. Устройство X10 — относительно рентабельное решение по визуальному оповещению всех участников проекта о состоянии построения.

Листинг 9.4 демонстрирует файл `config.xml`, настраивающий сервер CruiseControl на включение устройства X10 A2 и выключение устройства A3 с использованием компьютерного интерфейса CM17A на разъеме COM1 при успехе или сбое построения. Данный пример демонстрирует, как управлять двумя лампами (красной и зеленой) самодельного светофора[6].

Листинг 9.4. Файл `config.xml`, настраивающий сервер CruiseControl на переключение устройств X10

```
<publishers>
    <!-- Успешное построение: включить зеленую лампу -->
    <x10
        port="COM1"
        houseCode="A"
        deviceCode="2"
        onWhenBroken="false"
        interfaceModel="CM17A"/>
    <!-- Неудачное построение: включить красную лампу -->
    <x10
        port="COM1"
        houseCode="A"
        deviceCode="3"
        onWhenBroken="true"
        interfaceModel="CM17A"/>
</publishers>
```

По умолчанию сигнал "включения" подается на устройство при сбое построения, а сигнал "выключения" — при успехе. Если вам нужно противоположное поведение при успехе и провале, установите для атрибута `onWhenBroken` значение `false`. Сервер CI постоянно посылает сообщение устройствам X10, но если это такое же сообщение, как и прежде, лампа не изменит состояния.

В любое время в любом месте

Однажды во время завтрака на конференции в Денвере я получил на мобильный телефон текстовое сообщение о неудаче построения одного из наших проектов в Вирджинии. Я позвонил техническому руководителю проекта, и он проинформировал меня о характере проблемы. После поиска неисправности они выяснили, что неудачу потерпела одна из проверок JUnit, поскольку один из интерфейсов компонента изменился. Это продемонстрировало мне, как в действительности автоматизированная система построения, проверки, инспекции и развертывания программного обеспечения (а также других действий) позволяет узнавать о проблемах и участвовать в их устранении, даже находясь далеко от места работы и не имея доступа к электронной почте.

[6] Инструкция по эксплуатации находится по адресу *http://cruisecontrol.sourceforge.net/main/configxml.html#x10*.

Панель задач Windows

Система CCTray способна контролировать построение на сервере CruiseControl.NET и оповещать о его состоянии при помощи панели задач Windows. Таким образом, вам не нужно ждать сообщения и открывать электронную почту, чтобы узнать о состоянии построения, достаточно лишь взглянуть на пиктограмму, находящуюся на рабочем столе. Рис. 9.6 демонстрирует пиктограмму CCTray на панели задач Windows, а также состояние последнего построения (текст всплывающей подсказки). Установка CCTray осуществляется при установке сервера CruiseControl.NET. Ее установка и настройка довольно просты.

Рис. 9.6. Пиктограмма CCTray и сообщение о состоянии построения на панели задач Windows

Это весьма полезное средство, поскольку участникам группы достаточно посмотреть на панель задач Windows, чтобы выяснить состояние построения. Рассматривая реализацию сообщений на панели задач Windows в качестве механизма обратной связи, учитывайте следующее.

Требования: операционная система Windows и сервер CruiseControl.NET или Cruise Control.

Преимущества: ненавязчивая обратная связь в реальном масштабе времени.

Недостатки: доступно только для систем Windows.

Звуки

Звук — это еще один элемент, который может немного развлечь на рабочем месте и оказаться полезным, если вы находитесь в пределах слышимости. Рассматривая звук в качестве механизма обратной связи, учитывайте следующее.

Требования: звуковая плата и динамики.

Преимущества: способность оповестить многих людей одновременно и внести разнообразие в рабочий процесс.

Недостатки: как правило, звук воспроизводится только один раз. Вы должны находиться поблизости, чтобы его услышать. Если вы носите наушники (которые не подключены к компьютеру, проигрывающему звук), то вы можете не услышать его. Люди с проблемами слуха не воспримут такое оповещение.

Для уведомления о состоянии построения я люблю прибегать к различным звукам. Звук можно использовать совместно с другим устройством CFD, а также с системной панелью задач Windows и электронной почтой. Я использовал правила электронной почты, чтобы воспроизводить определенный звук в зависимости от темы электронной почты. Например, когда поступает сообщение о сбое построения, проигрывается звуковой фрагмент из мультфильма *Офисное пространство* (Office Space), говорящий: "У нас здесь проблема", а если построение успешно, проигрывается фраза из фильма *Аполлон 13* (Apollo 13): "Хьюстон, мы готовы к запуску". В некоторых из наших проектов группа разработки находится в той же комнате, что и машина построения. Используя те же звуки, машина построения объявляет об успехе или отказе при помощи динамиков. Листинг 9.5 демонстрирует сценарий делегирующего построения Ant, который поддерживает эти функциональные возможности. Данный сценарий делегирующего построения вызывается на сервере CruiseControl файлом конфигурации (`config.xml`).

Листинг 9.5. Звуки оповещения на сервере CruiseControl

```
<project name="project-delegating-build" default="run-cc-build">
    <target name="run-cc-build" depends="registerSounds ">
    ...
    </target>
    <property name="sounds.dir" location="PATH_TO_SOUNDS"/>
    <target name="registerSounds" if="use.sounds">
       <sound>
           <fail source="${sounds.dir}/failure/problemhere.wav"/>
           <success source="${sounds.dir}/success"/>
       </sound>
    </target>
</project>
```

Немного объясним этот код.

- Вы можете зарегистрировать звуки построения в файле `delegating-build.xml` и удостовериться, что задача `<sound>` вызывается в начале сценария.
- Элемент `<fail>` задает воспроизведение звукового файла из каталога звуков отказа построения.
- Элемент `<success>` задает проигрывание звукового файла успеха построения.
- Строку *PATH_TO_SOUNDS* следует заменить указанием вашего каталога звуков.
- Изменив значение свойства `use.sounds` в файле `config.xml` сервера CruiseControl, вы можете отключить использование звуков построения.

Повторюсь, мы действительно верим во внедрение устройств, звуков и стилей уведомления, которые вносят оживление и индивидуализируют непрерывную обратную связь. Мы полагаем, что эти устройства демонстрируют, насколько серьезно группа относится к своей работе, а не наоборот.

Широкоэкранные мониторы

Для повышения наглядности того, что ваша группа считает важным, вы можете использовать широкоэкранный монитор, основным преимуществом которого является автоматизация информации. Рассматривая возможность его применения в качестве механизма обратной связи, учитывайте следующее.

Требования: сетевое соединение и видеопроектор или широкоэкранный монитор.

Преимущества: автоматизированная, наглядная информация в реальном масштабе времени.

Недостатки: несколько опережающая зависимость цены от возможностей и типов автоматизируемой информации.

Для описания механизмов связи, которые “излучают” (radiate) информацию, Алистер Кокбурн (Alistair Cockburn) использовал термин *информационный излучатель* (information radiator). Когда он первоначально задумывал эту идею, ему нужна была некая большая установка, которую смогут видеть все находящиеся поблизости (это называлось *большой визуальной диаграммой* — Big Visual Chart — BVC). Первоначально для этого использовались цвета и большие буквы, но мы можем усовершенствовать этот способ технологически. BVC неэффективен для распределенных групп разработки, так как для обновления отображаемой информации они требуют скучной ручной работы. Поскольку теперь боль-

шую часть этой информации способна создавать система CI, вы можете использовать ее для управления отображением.

Невозможно сосчитать, сколько раз я слышал на работе диалоги, начинающиеся со слов: "Вы получили мою электронную почту?" или "На днях я проводил проверку последней версии файла в хранилище CVS", или "Когда следующая проверка проекта по расписанию?" Исходя из моего личного опыта связь — это одна из сложнейших проблем в проектах программного обеспечения. Как правило, проблема не в том, чтобы наладить связь, а в том, чтобы сделать это правильным способом.

Информационные излучатели делают расписание проекта, его показатели, результаты построения и другую информацию видимой всем участникам проекта, а модифицируется она автоматически. Когда люди читают эту информацию каждый день, они неизбежно ее осознают. Удостоверьтесь, что отображаемая информация о проекте совпадает с указываемой в индивидуальных сообщениях (по электронной почте или SMS), т.е. содержит некоторые ключевые данные, которые при необходимости обновляются. В противном случае широкоэкранный монитор окажется тем же морем информации, но в другой форме.

Дополнительные устройства обратной связи

Имеется и много других типов устройств и механизмов, которые вы можете использовать для связи; только сначала удостоверьтесь, что информация содержательна, кратка, своевременна и интересна. Главная задача — максимально быстро принять меры на основании информации. Информацию CFD время от времени необходимо обновлять, чтобы она не устарела. Существуют и другие идеи по поводу CFD, которые можно использовать в проектах.

- **Дополнение для браузера**. Существует весьма полезное дополнение Firefox[7] для Web-браузера, которое отображает состояние построения с использованием красных и зеленых индикаторов (как на панели задач Windows).
- **Мгновенный обмен сообщениями**. Оповещайте участников проекта о состоянии построения при помощи таких приложений мгновенного обмена, как AIM или Yahoo.
- **RSS**. Публикуйте результаты вашего построения, используя стандарт RSS (Really Simple Syndication). Файл XML стандарта RSS обновляется для каждого построения. Для получения этих обновлений вы можете использовать читатель вместо регулярной проверки электронной почты. Поддержку стандарта RSS обеспечивают многие серверы CI.
- **Элементы управления**. Для платформ Windows и Mac существуют разные элементы управления, позволяющие контролировать серверы CruiseControl и оповещать о состоянии построения.

Резюме

В этой главе вы изучили, как использовать возможности системы CI для автоматизации обратной связи на непрерывной основе. Передача правильной информации правильным людям в правильное время и правильным способом может решительно сократить промежуток времени между проявлением проблемы (или риска) и ее устранением. Это помогает улучшать качество программного обеспечения и снизить риск возникновения проблем.

[7] Более подробная информация о дополнении Firefox для сервера CruiseControl приведена по адресу `www.md.pp.ru/mozilla/cc/`.

Вопросы

Следующие вопросы помогут наладить механизм непрерывной обратной связи в среде разработки.

- Автоматизированы ли ваши процессы обратной связи?
- Включена ли ваша обратная связь в систему CI, чтобы не нужно было рассылать ее сообщения вручную?
- Те ли люди оповещаются? Не слишком ли много людей оповещаются и не слишком ли часто?
- Действительно ли обратная связь своевременна? Получают ли участники проекта сообщение сразу же после выявления проблемы?
- Получают ли участники проекта достаточный объем информации?
- Распределена ли ваша группа географически? Используете ли вы автоматизированные информационные излучатели?
- Делаете ли вы обратную связь интересной? Используете ли вы в процессах обратной связи такие средства, как звук или шар рассеянного света?

Эпилог

Будущее CI

Я заметил, что у тех, кто занимается CI некоторое время, обычно возникают два вопроса.

- Как предотвратить сбойные построения?
- Как ускорить процесс построения?

Здесь я рассмотрю каждый из этих вопросов, хотя не думаю, что мы найдем "совершенное" решение для них за столь короткое время.

Хотя практика CI обеспечивает более быструю обратную связь в малых интеграциях, это все еще довольно реакционная практика. Некоторые люди предпочитают осуществлять интеграцию вручную, поскольку хотят всегда поддерживать построение в успешном состоянии. Для удачного выполнения интеграции я предпочитаю использовать на отдельной машине больше вспомогательных инструментов, очереди, а также проверки перед передачей измененного исходного кода в хранилище с контролем версий.

Представьте, что единственное действие, которое должен осуществить разработчик, — это "передача" своего кода системе с контролем версий. Прежде чем хранилище примет код, оно запускает интеграционное построение на отдельной машине. И если оно проходит успешно, код будет передан в хранилище. Это может значительно уменьшить количество сбойных интеграционных построений и снизить потребность в их выполнении вручную. Во время публикации настоящей книги мы обнаружили прекрасный инструмент[1] для поддержки данного подхода, а в ближайшие годы ожидаем появления и других подобных инструментов.

Рис. Э.1 демонстрирует автоматизированное интеграционное построение с использованием очереди. Разработчик передает измененный код, и процесс прерывает запросы, чтобы начать передачу и запуск интеграционного построения на машине построения в очереди, чтобы гарантировать отсутствие конфликтов с другими переданными изменениями. При успешном завершении интеграционного построения код передается в хранилище. Если разработчик пытается передать код во время интеграционного построения, сервер поставит его в очередь, пока не закончится первое интеграционное построение.

Рисунок демонстрирует также более эффективную систему с контролем версий, поддерживающую средства CI. Кажется вполне логичным, что поскольку система с контролем версий всегда запущена и в ней нуждается эффективная система CI, вы могли бы использовать ее для предотвращения передачи сбойного кода, проверки или даже инспекции всего передаваемого в совместно используемую базу кода.

---

[1] См. Gauntlet от Borland (`www.borland.com/us/products/silk/gauntlet/`), TeamCity от JetBrains (`www.jetbrains.com/teamcity/`) и Team Foundation Server от Microsoft (TFS) (`http://msdn2.microsoft.com/en-us/teamsystem/`). На момент публикации корпорация Microsoft не предоставляла дополнительной поддержки для CI, вместо этого использовалось построение по расписанию.

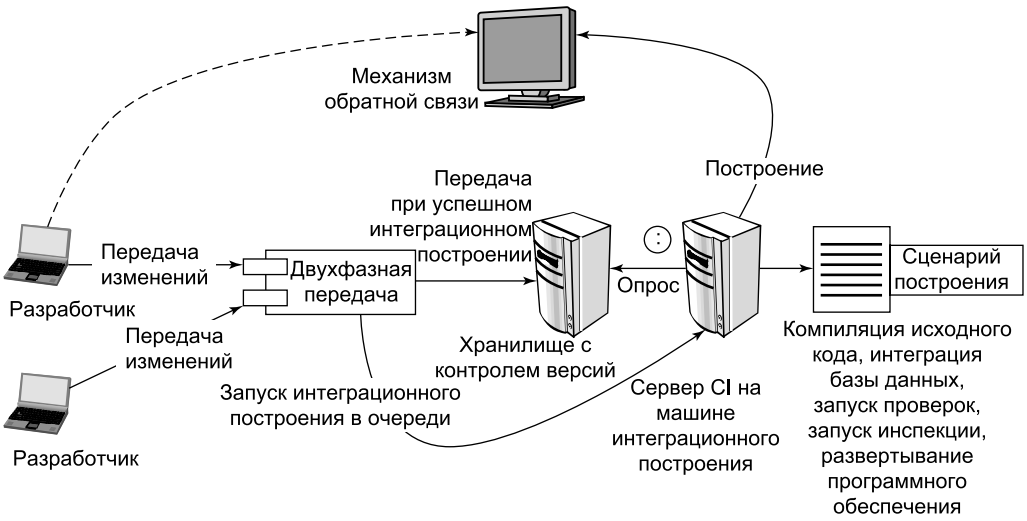


Рис. Э.1. Будущее CI — автоматизированная постановка в очередь интеграционного построения

Альтернативный подход предотвращения сбойных построений подразумевает наличие у разработчика возможности запустить интеграционное построение *на машине интеграционного построения с его локальными изменениями* (которые не были переданы в хранилище с контролем версий) наряду со всеми другими изменениями, переданными в хранилище с контролем версий[2]. Когда такая методика поддерживается всеми разработчиками, она может существенно снизить количество сбойных построений, поскольку вы интегрируете все ваши изменения и запускаете интеграционное построение на отдельной машине перед их передачей в хранилище.

Еще одна область усовершенствования CI — это обеспечение более быстрой обратной связи и более быстрого построения. В главе 4 рассматриваются методы и возможные решения, но я ожидаю появления большего количества средств в областях параллельной обработки и других возможностей манипулирования дополнительными аппаратными средствами и ресурсами, а также программного обеспечения, позволяющего ускорить построение.

[2] Зутуби (Zutubi) назвал это “персональным построением”, оно поддерживается сервером CI Pulse.

Приложение A
Ресурсы CI

Это приложение предоставляет информацию об инструментальных средствах и ресурсах CI, категоризированную по следующими темам:

- Web-сайты и статьи по непрерывной интеграции;
- инструменты CI и производственные ресурсы;
- ресурсы по сценариям построения;
- ресурсы по системам с контролем версий;
- ресурсы по базам данных;
- ресурсы по проверке;
- ресурсы по автоматизированной инспекции;
- ресурсы по развертыванию;
- ресурсы по обратной связи;
- ресурсы по документированию.

Web-сайты и статьи по непрерывной интеграции

Автоматизация для людей: непрерывная обратная связь

- `http://www-128.ibm.com/developerworks/java/library/j-ap11146/`

Эта статья на IBM developerWorks рассматривает различные механизмы обратной связи, применяющиеся в среде CI.

Автоматизация для людей: непрерывная инспекция

- `http://www-128.ibm.com/developerworks/java/library/j-ap08016/`

В этой статье на IBM developerWorks рассматриваются такие автоматизированные инспекторы, как Checkstyle, JavaNCSS и CPD, способы усовершенствования процесса разработки и случаи их применения.

Автоматизация для людей: улучшение сценариев построения

- `http://www-128.ibm.com/developerworks/java/library/j-ap10106/`

Эта статья на IBM developerWorks анализирует способы усовершенствования сценариев построения на примере среды Ant.

Непрерывная интеграция

- `www.martinfowler.com/articles/continuousIntegration.html`

Мартин Фаулер представляет принципы и практики CI.

Непрерывная интеграция

- `www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&id=58`

Статья Джеффри Фредерика по поводу CI в журнале *Better Software Magazine*.

Ежедневное построение и бездымовое тестирование

- `www.stevemcconnell.com/bp04.htm`

Чтобы никто не подумал, что практика CI очень нова или появилась ниоткуда, авторы представляют еще одного весьма влиятельного лидера программного обеспечения, Стива Мак-Коннелла (Steve McConnell), обсуждающего ежедневное построение и бездымовые тестирование в *IEEE Software*, том 13, номер 4, июль 1996 года.

IntegrateButton.com

- `www.integratebutton.com`

Это Web-сайт одного из авторов книги, посвященный информации и исследованиям в области CI, включающий форум, примеры и многое другое.

Реализация непрерывной интеграции

- `http://www-128.ibm.com/developerworks/rational/library/sep05/lee/`

Статья Кевина Ли (Kevin Lee) на IBM developerWorks, представляющая концепцию и практики CI.

Инструменты CI и производственные ресурсы

AnthillPro

- `www.urbancode.com/products/anthillpro/`

Коммерческий сервер управления построением, предоставляющий возможности CI. См. также приложение Б, “Обсуждение инструментальных средств CI”.

Apache Continuum

- `http://maven.apache.org/continuum/`

Web-сайт проекта Apache Maven. См. также приложение Б, “Обсуждение инструментальных средств CI”.

Bamboo

- `www.atlassian.com/software/bamboo/`

Коммерческий сервер CI, бесплатно доступный для проектов с открытым исходным кодом. Сервер Bamboo предоставляет показатели построения, обладает простым UI и интеграцией с такими инструментами Atlassian, как JIRA.

BuildForge

- `http://www-306.ibm.com/software/awdtools/buildforge/enterprise/`

BuildForge — высокопродуктивный коммерческий инструмент управления построением, предоставляющий мощнейшие функциональные возможности по распределенному построению, проверке и развертыванию.

Матрица серверов непрерывной интеграции

- `http://damagecontrol.codehaus.org/Continuous+Integration+Server+Feature+Matrix`

Эта матрица предоставляет краткий обзор рынка коммерческих и бесплатных (с открытым исходным кодом) серверов CI. Здесь описывается много критериев, позволяющих выбрать наиболее подходящий сервер.

CruiseControl

- `http://cruisecontrol.sourceforge.net`

Сервер CruiseControl написан полностью на Java и является одним из первых серверов CI (доступен начиная с 2001 года). См. также приложение Б.

CruiseControl.NET

- `http://ccnet.thoughtworks.com`

Написанный на языке C#, этот сервер CI основан на Java-версии сервера CruiseControl. Он также является реализацией с открытым исходным кодом и доступен бесплатно. См. также приложение Б.

Draco.NET

- `http://draconet.sourceforge.net/`

Бесплатный сервер CI с открытым исходным кодом. См. также приложение Б.

Gauntlet

- `www.borland.com/us/products/silk/gauntlet/`

Guantlet поддерживает возможность, называемую “sandboxing”, которая изолирует изменения исходного кода, пока интеграционное построение не пройдет успешно. Сервер TeamCity от JetBrains предоставляет подобную возможность, являющуюся следующим шагом развития CI, поскольку это может предотвратить неудачу построения в связи с передачей в хранилище с контролем версий сбойного кода.

Luntbuild

- `http://luntbuild.javaforge.com/`

Luntbuild — сервер управления построением, поддерживающий также функции CI. См. также приложение Б.

ParaBuild

- `www.viewtier.com/products/parabuild/index.htm`

ParaBuild — коммерческий, автоматизированный сервер управления построением.

QuickBuild от PMEase

- www.pmease.com/

QuickBuild — профессиональная версия сервера Luntbuild.

Sin

- http://sin.tigris.org/

Sin (официальное название — Continuous Integration for Subversion) — это средство, помогающее CI предотвращать повреждение хранилища с контролем версий за счет использования подхода защиты "checkin branches" (проверка ветвей), подразумевающего проверку исправлений перед их внедрением в основную линию (т.е. слиянием). Для Sin требуется платформа .NET и хранилище Subversion.

Другие инструменты CI и производственные ресурсы

Название	Web-сайт
Bitten	http://bitten.cmlenz.net/
BuildBeat	www.timpanisoftware.com/
BuildBot	http://buildbot.sourceforge.net/
CM Crossroads	www.cmcrossroads.com/
CruiseControl.rb	http://cruisecontrolrb.thoughtworks.com/
Gump	http://gump.apache.org/
PerfectBuild	www.codefast.com
Pragmatic Automation	www.pragmaticautomation.com/
Pulse	www.zutubi.com/products/pulse/
TeamCity	www.jetbrains.com/teamcity/
Tinderbox	www.mozilla.org/tinderbox.html

Ресурсы по сценариям построения

Ant

- http://ant.apache.org

Ant, вероятно, наиболее популярный инструмент сценариев построения для групп разработки Java. Если вы участвуете в проекте Java, имеет смысл потратить время на изучение этого средства. Большинство инструментов CI поддерживают Ant.

Groovy

- http://groovy.codehaus.org/
- www.javaworld.com/javaworld/jw-10-2004/jw-1004-groovy_p.html
- http://www-128.ibm.com/developerworks/library/j-pg12144.html

Groovy — это динамический язык для платформы Java, который можно использовать для написания сценариев Ant XML. Применяя программные конструкции Groovy, вы можете писать сценарий для вашего процесса построения.

Maven

- `http://maven.apache.org`

Инструмент управления проектом и построением. См. также приложение Б.

NAnt

- `http://nant.sourceforge.net/`

NAnt — порт Java-ориентированного инструмента Ant для платформы .NET.

Rake

- `http://rake.rubyforge.org/`

Rake — инструмент сценариев построения для Ruby-ориентированных приложений. Использование инструмента Rake позволяет также задействовать всю мощь Ruby в сценариях построения.

Ресурсы по системам с контролем версий

ClearCase

- `www.ibm.com/software/awdtools/clearcase/`

Коммерческий инструмент управления конфигурацией программного обеспечения с многочисленными дополнительными возможностями.

Concurrent Versions System (CVS)

- `www.nongnu.org/cvs/`

Инструмент контроля версий с открытым исходным кодом для платформы UNIX.

MKS

- `www.mks.com/`

Коммерческая версия инструмента контроля.

Subversion

- `http://subversion.tigris.org/`

Следующее поколение популярного инструмента контроля версий CVS с открытым исходным кодом.

Другие ресурсы по системам с контролем версий

Название	Web-сайт
AccuRev	`www.accurev.com/`
Alienbrain	`www.alienbrain.com/`
Perforce	`www.perforce.com/`
PVCS	`www.serena.com/Products/professional/vm/home.asp`
SnapshotCM	`www.truebluesoftware.com/`
StarTeam	`www.borland.com/us/products/starteam/index.html`

Название	Web-сайт
Surround SCM	`www.seapine.com/surroundscm.html`
Synergy CM	`www.telelogic.com/corp/products/synergy/index.cfm`
Visual SourceSafe	`http://msdn.microsoft.com/vstudio/Previous/ssafe/default.aspx`

Ресурсы по базам данных

Hypersonic DB

- `www.hsqldb.org/`

HSQLDB — облегченная (порядка 100 Кбайт) база данных, написанная полностью на Java и доступная бесплатно. Прекрасно подходит для управления проверочными данными приложения в ходе проверки разработчика.

Mckoi

- `www.mckoi.com/database/`

Mckoi — еще одна реализация облегченной базы данных SQL для Java, с открытым исходным кодом (по лицензии GPL). Прекрасно подходит для групп, желающих использовать при разработке "пространство базы данных разработчика". Для приведения вашего кода SQL в соответствие с кодом Sybase и Oracle понадобится некоторая работа, но это вполне реализуемо.

MySQL

- `www.mysql.com`

MySQL представляет собой комплект мощнейших баз данных, которые первоначально являлись системой реляционных баз данных с открытым исходным кодом, способных работать на всех основных операционных системах, включая Linux, UNIX и Windows. Сегодня система MySQL превратилась в лидера отрасли, успешно конкурирующего с именитыми производителями. Издание Community Edition доступно бесплатно согласно лицензии GPL.

Oracle

- `www.oracle.com/technology/database/index.html`

Известная система управления реляционными базами данных корпоративного класса, работающая на всех основных операционных системах, включая Linux и Windows. Издание Oracle Express Edition наилучшим образом подходит для разработчиков: оно бесплатно для разгрузки, разработки, развертывания и распространения, а кроме того, это облегченная версия серии продуктов Oracle, включающей издания Standard и Enterprise Editions.

PostgreSQL

- `www.postgresql.org/`

PostgreSQL — мощнейшая система реляционной базы данных с открытым исходным кодом, работающая на всех основных операционных системах, включая Linux, UNIX (AIX, BSD, HP-UX, Mac OS X, SGI IRIX, Solaris и Tru64), а также Windows.

Ресурсы по проверке

Agitator

- `www.agitar.com/products/`

AgitarOne от Agitar — это коммерческий продукт, который автоматически создает набор проверок для кода Java.

DbUnit

- `http://dbunit.sourceforge.net`

DbUnit — расширение JUnit с открытым исходным кодом, возвращающее базу данных в исходное состояние между проверками.

Fit

- `http://fit.c2.com/`

Fit — инструмент с открытым исходным кодом, облегчающий связь между деловыми клиентами, предъявляющими требования, и разработчиками, реализующими их. Доступен для Java, .NET, Ruby и Python.

FitNesse

- `http://fitnesse.org/`

FitNesse — инструмент с открытым исходным кодом, обеспечивающий проверку Fit при помощи вики. Инструмент FitNesse доступен для .NET, Java и Ruby.

Floyd

- `www.openqa.org/floyd/`

Floyd — инструмент проверки с открытым исходным кодом для платформы Java, имитирующий браузер при проверке Web-ориентированных приложений.

HtmlUnit

- `http://htmlunit.sourceforge.net/`

HtmlUnit — инструмент Java с открытым исходным кодом, контролирующий среду выполнения перед проверкой Web-ориентированных приложений.

JUnit

- `http://junit.org`

JUnit — среда проверки модулей с открытым исходным кодом для Java.

JWebUnit

- `http://jwebunit.sourceforge.net/`

JWebUnit — среда выполнения Java с открытым исходным кодом, облегчающая разработку приемочных испытаний для Web-приложений.

NDbUnit

- `www.ndbunit.org/`

NDbUnit — библиотека .NET с открытым исходным кодом, применяемая для возвращения базы данных в заданное состояние. NDbUnit используется для улучшения повторяемости проверок, в которых задействованы базы данных, за счет обеспечения однозначности состояния базы данных при каждой проверке.

NUnit

- `www.nunit.org/`

NUnit — среда проверки модулей с открытым исходным кодом для всех языков платформы .NET.

Selenium

- `www.openqa.org/selenium`

Selenium — инструмент проверки функций Web-приложений в стиле Fit (набор тестов в виде таблицы), т.е. "в окне просмотра". Прекрасно подходит для групп разработки, желающих автоматизировать регрессивные проверки системы своих Web-приложений. Легко интегрируется в систему CI. Прекрасный вспомогательный инструмент с открытым исходным кодом, IDE Selenium облегчает создание сценариев проверки, позволяя испытателям записывать свои действия при использовании приложений (необходимы некоторые базовые знания HTML и JavaScript).

SQLUnit

- `http://sqlunit.sourceforge.net`

SQLUnit — среда проверки хранимых процедур базы данных с открытым исходным кодом.

TestEarly.com

- `www.testearly.com/`

TestEarly.com — блог, посвященный качеству построения программного обеспечения на раннем этапе цикла разработки. Некоторые из авторов этой книги — постоянные участники дискуссий, проводимых на этом сайте.

TestNG

- `www.testng.org`

TestNG — среда проверки с открытым исходным кодом для платформы Java. Созданная под впечатлением JUnit и NUnit, представляет собой некое новое средство, прекрасно подходящее для проверки на уровне компонентов и системы.

utPLSQL

- `http://utplsql.sourceforge.net/`

utPLSQL — среда проверки с открытым исходным кодом, применяемая для исследования программ на языках PL и SQL для Oracle.

Watir

- `www.openqa.org/watir`

Watir — инструмент проверки функций с открытым исходным кодом, написанный на языке Ruby. Предназначен для автоматизации браузер-ориентированных проверок Web-приложений.

Шаблоны проверки xUnit

- `http://xunitpatterns.com/`

Web-сайт книги Джерарда Месзароса (Gerard Meszaros) *xUnit Test Patterns*.

Ресурсы по автоматизированной инспекции

Checkstyle

- `http://checkstyle.sourceforge.net`

Checkstyle — инструмент инспекции и контроля за соблюдением стандартов программирования для кода на языке Java. Начиная с версии 3, возможны проверки не только на типичные случаи несоблюдения стандартов программирования. В настоящее время Checkstyle поддерживает различные типы проверок и инспекций, включая проверку проекта, сложности и дублирования кода.

Clover

- `www.cenqua.com/clover/`

Clover — коммерческий инструмент исследования покрытия кода Java и .NET.

Cobertura

- `http://cobertura.sourceforge.net/`

Cobertura — инструмент исследования покрытия с открытым исходным кодом для Java.

EMMA

- `http://emma.sourceforge.net/`

EMMA — инструмент исследования покрытия с открытым исходным кодом для Java. Отчеты EMMA несколько отличаются от отчетов Cobertura.

FindBugs

- `http://findbugs.sourceforge.net/`

FindBugs — Java-ориентированный инструмент инспекции и поиска ошибок в коде Java на основании шаблона ошибки. Включите этот инструмент в процесс построения и создавайте отчеты. Вы будете удивлены тем, как много еще не знали о программировании на языке Java.

FxCop

- `www.gotdotnet.com/Team/FxCop/`

FxCop — инструмент анализа кода для платформы .NET, который анализирует сборки на предмет соответствия принципам .NET Framework Design Guidelines.

JavaNCSS

- `www.kclee.de/clemens/java/javancss/`

JavaNCSS — инструмент с открытым исходным кодом, позволяющий определять длину методов и классов, а также исследовать файлы исходного кода Java.

JDepend

- `www.clarkware.com/software/JDepend.html`

JDepend просматривает файлы класса Java и выявляет показатели качества для каждого пакета.

NCover

- `http://ncover.org`

NCover — инструмент покрытия с открытым исходным кодом для платформы .NET.

NDepend

- `www.ndepend.com/`

NDepend анализирует код .NET и выявляет показатели качества проекта, включая степень центростремительной и центробежной связи, неустойчивость и ряд других интересных показателей.

PMD

- `http://pmd.sourceforge.net`

PMD — статический анализатор кода для платформы Java (с открытым исходным кодом).

Simian

- `www.redhillconsulting.com.au/products/simian/`

Simian — инструмент выявления дублирования кода Java, C#, C++, Ruby и некоторых других современных языков. Способен искать повторения в текстовых файлах.

SourceMonitor

- `www.campwoodsw.com/sm20.html`

SourceMonitor — инструмент инспекции (сбора показателей), бесплатный для программистов. Поддерживаются языки программирования C/C++, Delphi, HTML, Java, C# и Visual Basic. Проанализируйте код и найдите способы его улучшения; если вам непонятно, что означают некоторые показатели, вы можете обратиться к документации, описывающей их. Вы можете преобразовать отчеты XML в формат HTML, чтобы их можно было включать в процесс построения.

Ресурсы по развертыванию

Capistrano (ранее SwitchTower)

- `http://manuals.rubyonrails.com/read/book/17`

Capistrano — утилита для развертывания Web-приложений на программном каркасе Ruby on Rails.

Ресурсы по обратной связи

Ambient Devices

- `www.ambientdevices.com/`
- `www.qualitylabs.org/projects/ambientorb/`

Компания Ambient Devices предлагает множество товаров. В главе 9 упомянуто использование шара рассеянного света в качестве "наглядного" информационного излучателя. Задача Ant, упрощающая взаимодействие с шаром рассеянного света, доступна у Quality Labs.

GoogleTalk

- `www.google.com/talk/`

При некоторых работах вы можете задействовать сообщения Jabber, которые система CI будет посылать (например, CruiseControl) вашему клиенту при помощи службы мгновенного обмена сообщениями.

Jabber

- `www.jabber.org/`

Система мгновенного обмена сообщения с открытым исходным кодом, включаемая в систему обратной связи CI. Система Jabber совместима с системой GoogleTalk.

X10

- `www.x10.com/`

Вы можете использовать стандарт X10 для управления любыми электротехническими устройствами, которые используют радиочастоту. Этот сайт содержит информацию о начальном комплекте, который можно применить для оживления рабочей обстановки и усовершенствования механизмов обратной связи организации.

Другие ресурсы

Название	Web-сайт
Корпоративный почтовый сервер Apache Java ("Apache James")	`http://james.apache.org/server/index.html`
Gaim	`http://gaim.sourceforge.net/`
Лава-лампы[1]	`www.lavalites.com/`

[1] См. `http://www.utx.ru/goods.php?id=2350`. — *Примеч. ред.*

Ресурсы по документированию

Doxygen

- `www.stack.nl/~dimitri/doxygen/`

Doxygen — система документирования с открытым исходным кодом для языков C/C++, Java, Objective-C, Python, IDL (Corba и разновидности Microsoft), а также в меньшей степени для PHP, C# и D. Эта программа позволяет создавать документацию в различных форматах, включая LaTeX, RTF, PostScript, PDF, HTML и man page для UNIX. Вероятно, наилучшей возможностью системы Doxygen является средство GraphViz, обеспечивающее построение диаграмм в стиле UML, помогающих визуализировать исходный код.

Javadoc

- `http://java.sun.com/j2se/javadoc/`

Платформа Java обладает стандартным инструментом создания документации API в формате HTML. Существуют различные "доклеты" ("doclet"), позволяющие получать документацию в разных форматах, а также проверять комментарии Javadoc.

NDoc

- `http://ndoc.sourceforge.net/`

NDoc — инструмент документации с открытым исходным кодом для платформы .NET (а именно, сборок .NET и файлов документации XML, предназначенных для языка C#). Этот инструмент поможет создать документацию способом, стандартным для Microsoft, например `.chm`, HTML Help 2 и MSDN в стиле Web-страниц.

Приложение Б

Обсуждение инструментальных средств CI

Мастер тот, кто перед работой готовит свои инструменты.

Китайская пословица

Рауль (это ненастоящее имя) входит в состав небольшой группы, решающей вопросы проекта J2EE для большой группы разработки. Его роль в этом предприятии — ведущий интегратор: он несет ответственность за обеспечение совместимости друг с другом порядка шестидесяти средств разработки, а также средств проверки, рабочих сред и сред построения. Его основной задачей является сбор исходного кода и других артефактов построения, используемых для создания среды разработки. Ему необходимо подобрать хранилище с контролем версий и файловую систему с сетевой структурой. И тут один из участников группы очень кстати замечает: "По-моему, у Карла в отделе есть на дискете довольно хороший экземпляр конфигурации сервера приложений".

Имея исходные артефакты на руках, следующая задача Рауля заключается в создании автоматизированного процесса построения для сред разработки. Рабочие процессы проверки и построения стабильны, но представляют собой набор сценариев оболочки UNIX, которые проверяют код, вызывают компиляторы, копируют файлы JAR и т.д., но, к сожалению, все среды разработки предназначены для машин Windows, каждая из которых имеет JVM, экземпляр сервера приложений и инструменты редактирования, установленные разработчиком.

Расстроенный вероятной потерей производительности участниками проекта во время интеграции, Рауль обратился к менеджеру по конфигурации проекта. "Мы уже дошли до предела, — сообщил Рауль. — Мы собираемся раздобыть и установить на рабочих станциях программное обеспечение эмуляции UNIX, чтобы разработчики могли запускать сценарии построения UNIX". С учетом больших работ по перенастройке и такого же количества несогласных этот ненадежный подход, вероятно, сработает, по крайней мере только для тех разработчиков, которые настроили свои рабочие станции сходно с окружением UNIX.

Рауль, обладая некоторыми политическими и риторическими навыками (в чем он большой мастер, по его словам), убедил руководителя проекта изменить курс в сторону механизма построения на базе Ant. Механизмы Ant и Java независимы от платформы, а кроме того, сценарии Ant можно использовать даже для автоматической установки среды разработки, сэкономив разработчикам много рабочих часов.

Мораль сей истории такова: выбор инструментов — непростое дело. На самом деле существует несколько способов решения любой задачи, однако некоторые из них проще и удобней других. К счастью, отдавая предпочтение любому подходу кроме "своего собственного", вы избежите крупных ошибок при выборе инструментов реализации среды разработки. Большинство общедоступных инструментов хорошо проработаны и прекрасно подходят для решения задач CI.

Это приложение поможет вам определиться с выбором соответствующих инструментов CI. К сожалению, я не могу порекомендовать вам средство, совершенное во всех отношениях, поскольку выбор такового зависит от вашей среды, размера проекта и функций, которые необходимы при автоматизированном построении. Какой инструмент лучше для забивания гвоздей — молоток или гвоздобойный пистолет? Вариантов ответа будет два, в зависимости от решаемой задачи: предстоит ли построить гараж или скворечник. Именно таким аспектам, которые следует учитывать при выборе инструментов для реализации CI вашей группой разработки, и посвящен первый раздел данного приложения.

Второй и третий разделы содержат краткий обзор средств, доступных в настоящее время. Хотя я и не привожу подробных инструкций их применения, информации будет достаточно, чтобы получить общее представление о самих инструментах, их установке и использовании. Я рассматриваю здесь только те инструменты, которые подходят для двух наиболее популярных платформ разработки приложения: Java и .NET. Если вы работаете на другом языке, например типа Ruby, C, Perl или PHP, не впадайте в отчаяние, существуют средства CI для широкого диапазона стилей разработки и языков. Поиск в Интернете позволит быстро подобрать все необходимое для этих платформ.

Обновляйте это приложение

Поскольку описанные в данном приложении инструменты быстро меняются, мы рекомендуем посетить Web-сайт книги, `www.integratebutton.com`, чтобы получить наиболее свежую информацию по сценариям, инструментам и исследованиям.

Данное приложение рассматривает лишь инструменты построения и планирования, но не инструменты контроля версий, поскольку систему с контролем версий вам следует выбрать самостоятельно. В этом вам помогут множество прекрасных специализированных книг и сетевых ресурсов. Если вы уже участвуете в проекте, где не используется инструмент контроля версий, поставьте эту книгу на место и возьмите другую. Итак, приступим.

Аргументы при оценке инструментальных средств

Выбор программного обеспечения автоматизации — это вопрос поиска наиболее подходящих средств для вашей среды и процесса разработки. Наилучший инструмент тот, который сэкономит вам и остальной части группы больше времени и средств, а также прослужит дольше. Дискуссии при сравнении инструментов зачастую выходят за рамки практического смысла и превращаются в перепалку наподобие религиозного диспута. Иногда читаешь обсуждения о сравнении качеств серверов CruiseControl и Anthill и невольно вспоминаешь бесконечные дебаты между Фордом и Шевроле (хотя я никогда не видел никаких оскорбительных карикатур по поводу программного обеспечения).

Имейте также в виду, что инструменты вы выбираете не навечно. Когда инструмент задерживается надолго, значит, он подходит вам (а вы ему). В ходе пары проектов, над кото-

рыми я работал, пришлось менять инструмент построения и планирования прямо посреди переправы. И в обоих случаях это оправдалось. Конечно, если вы вложили существенные усилия и деньги в одно из сверхнадежных средств распределенного построения, то это совсем иное дело. Тем не менее для большинства из нас прекрасно подойдет один из многих инструментов с открытым исходным кодом.

Давайте рассматривать различные факторы, которые будут влиять на принимаемое вами решение. Их следует учитывать при выборе используемой среды разработки. Напомним, что это не то решение, которое потребует недель исследования и не будет подлежать пересмотру. После пары дней пробных непрерывных интеграций вы можете внести изменения и продолжить.

Функциональные возможности

Безусловно, важнейшим критерием при выборе инструмента является его функциональность. В этом разделе описаны основные и дополнительные функциональные возможности, предоставляемые инструментами построения и планирования.

Инструменты построения — основные функциональные возможности

Ниже приведены основные функциональные возможности инструментов построения.

- **Компиляция кода**. Здесь никаких сюрпризов: компиляция исходного кода — это основной компонент построения программного обеспечения. Для повышения эффективности она должна осуществляться при условии изменения либо исходного кода, либо зависимостей.
- **Упаковка компонентов**. После компиляции исходного кода и оформления любых других артефактов, которые должны быть включены в программное обеспечение, их обычно необходимо связать в развертываемые компоненты, такие как файлы JAR для Java или файлы EXE для Windows. Инструмент построения, который вы выберете, должен уметь добавлять в пакет необходимые компоненты вашей системы, и делать это только при изменении содержимого пакета.
- **Выполнение программы**. Инструмент построения должен обладать хорошими возможностями по запуску программ на целевой платформе, а также вызова любых программ, которые имеют интерфейс командной строки.
- **Манипулирование файлами**. Функциональные возможности создания, копирования, а также удаления файлов и каталогов присущи практически всем инструментам построения.

Инструменты построения — дополнительные функциональные возможности

К дополнительным функциональным возможностям инструментов построения относятся следующие.

- **Выполнение проверок разработчика**. Кроме обычной компиляции программного обеспечения, требуется также возможность запуска комплекта автоматизированных проверок разработчика. Хотя в случае необходимости инструмент построения с инструментом проверки вполне возможно интегрировать при помощи командной строки, лучше иметь инструмент построения, укомплектованный средством проверки модулей.

- **Интеграция инструментов контроля версий**. Если ваш инструмент планирования построений делегирует действия по контролю версий инструменту построения или если вам необходимы некие другие действия по контролю версий, связанные с автоматизацией, то желательно подобрать систему с контролем версий, имеющую внутренний инструмент построения. Напомним, что в случае необходимости вы всегда можете прибегнуть в качестве аварийного варианта к интеграции инструментов при помощи командной строки.
- **Создание документации**. Если вы работаете с языком программирования, который имеет встроенные средства документирования, например C# или Java, то очень полезно иметь инструмент построения, обладающий API автоматического создания документации в ходе построения.
- **Функциональные возможности развертывания**. Если при автоматизированном построении вы планируете запускать проверки функций или модулей "в контейнере", то построенное приложение необходимо сначала развернуть на проверочном сервере. Эти функциональные возможности могут быть предоставлены инструментом построения или производителем сервера (либо сообществом пользователей сервера) в качестве дополнения.
- **Анализ качества кода**. Как упоминалось в главе 7, прекрасное представление о стабильности и ремонтопригодности кода можно получить в результате запуска различных типов автоматизированных инспекторов. Выясните, какие средства анализа встроены в ваш инструмент построения или доступны как дополнения.
- **Расширяемость**. Как правило, писать собственные дополнения для инструмента построения не приходится; большинство проблем, с которыми вы столкнетесь, типичны и кем-то уже решались. Но в некоторых случаях вы можете захотеть усовершенствовать собственно инструмент построения; например, при желании полностью интегрировать новую проверку или средство составления отчетов. Хорошо задокументированные API расширения окажутся в этом случае весьма полезны. Только не забудьте пожертвовать свое дополнение сообществу пользователей. Вы, ваше дополнение и сообщество существенно выиграют от этого.
- **Многоплатформенное построение**. Большинство серверов CI разработано так, чтобы выполняться на одной машине построения. Это, безусловно, означает, что все действия построения будут происходить на платформе сервера построения. Для большинства приложений это подойдет. Но если вы разрабатываете программное обеспечение для нескольких платформ, все значительно усложняется. Наилучшим выходом в данном случае может стать приобретение одного из коммерческих инструментов, способных осуществлять процесс построения на нескольких серверах.
- **Ускоренное построение**. Ключевым преимуществом CI является возможность быстрого запуска полного цикла построения. По этой причине некоторые эксперты советуют удерживать полный цикл построения в пределах десяти минут[1]. Если же ваш цикл построения занимает несколько часов в связи с большим объемом кода (хотя это редкость), вы можете рассмотреть возможность применения некоторых инструментальных средств, которые способны распределить этапы построения на несколько процессов, выполняемых разными серверами построения.

[1] Кент Бек (Kent Beck). *Extreme Programming Explained, Second Edition.*

Инструменты планирования построения — основные функциональные возможности

Ниже приведены основные функциональные возможности инструментов планирования построения.

- **Запуск построения**. Основная функция планировщика построения — периодический запуск автоматизированного процесса построения. При этом существуют некоторые различия между тем, как те или иные инструменты определяют сам момент запуска. Некоторые инструменты используют для этого опрос. Они периодически опрашивают хранилище с контролем версий (обычно каждые несколько минут) и, обнаружив в нем изменение, запускают построение. Другие инструменты руководствуются расписанием. Они проверяют хранилище с контролем версий по заранее заданному расписанию или через определенные промежутки времени. Ортодоксы CI утверждают, что инструменты, управляемые расписанием, не являются настоящими серверами CI, поскольку их зачастую настраивают на ежедневное построение, не предусматривающее перенастройки на более короткие интервалы. На мой взгляд, с технической точки зрения они правы, поэтому лично я предпочитаю средства, управляемые опросом, но не забываю, что наилучшим инструментом является тот, который поможет решить задачу наиболее эффективно. Если вы считаете, что вам комфортнее работать при ежечасных построениях, так и поступайте, однако по определению это не будет считаться непрерывной интеграцией.
- И наконец, некоторыми инструментами управляют события. Это означает, что построение запускается автоматически, когда происходит событие изменения артефакта в системе с контролем версий. Хотя данный подход может показаться предпочтительней, он имеет незначительное практическое отличие от подхода с использованием опроса. Кроме того, управляемый событиями инструмент почти наверняка потребует некоторой перенастройки системы с контролем версий, в то время как инструмент, управляемый опросом, — нет.
- **Интеграция с системой контроля версий**. Безусловно, выбор инструмента, интегрированного с вашей системой контроля версий, крайне важен. Большинство инструментов поддерживают наиболее популярные системы с контролем версий, поэтому маловероятно, что вы не найдете для себя что-либо подходящее. Однако вам следует обратить внимание на то, как инструмент взаимодействует с вашей системой. Всегда ли он задействует полный набор файлов при каждом изменении, есть ли у него параметры, позволяющие задать такой режим построения? Этот подход может оказаться неприемлемым в большом проекте, и тогда вы попытаетесь перейти на почти непрерывное построение. Еще одна полезная возможность, которую следует учесть, — это то, насколько хорошо инструмент идентифицирует изменения файлов, входящих в построение. Инструмент должен как минимум выявлять измененные файлы и номера их версий.
- **Интеграция инструмента построения**. Еще один компонент, который следует учесть при интеграции с системой контроля версий. Большинство инструментов поддерживают наиболее популярные системы с контролем версий. Только удостоверьтесь, что инструмент взаимодействует с вашей версией платформы управления.
- **Обратная связь**. Обратная связь очень важна для CI. Все инструменты, перечисленные в данном приложении, поддерживают ее хотя бы по электронной почте, что может быть вполне достаточным, если не учитывать другие возможности, которые

могли бы вам пригодиться, например обратную связь с использованием мгновенных сообщений, текстовых сообщений или некоторых других устройств. Более подробная информация о некоторых устройствах обратной связи приведена в главе 9.

- **Маркировка построений**. В большинстве случаев вам понадобится инструмент, способный маркировать артефакты, которые использовались при данном построении. Это называется пометкой или маркировкой, в зависимости от вашей системы с контролем версий. Большинство инструментов поддерживает некую разновидность счетчика, который встроен в формат метки.

Инструменты планирования построения — дополнительные функциональные возможности

К дополнительным функциональным возможностям инструментов планирования построения относятся следующие.

- **Зависимости между проектами**. В соответствии с вашей стратегией управления конфигурацией, при наличии зависимости между проектами можно выполнять построение зависимого проекта одновременно с построением основного.
- **Пользовательский интерфейс**. Строго говоря, особой причины требовать наличия пользовательского интерфейса для инструмента планирования построения нет. Основные функциональные возможности выполняются как демон, проверяющий изменения в системе контроля версий и запускающий построение, посылая сообщение обратной связи. Тем не менее весьма полезно иметь пользовательский интерфейс, который позволяет изменять конфигурацию, проверять текущее состояние построения и загружать артефакты. Все инструменты обеспечивают это в той или иной форме, обычно в качестве интерфейса Web-приложения. Некоторые инструменты, такие как Luntbuild, распространяются как Web-приложения. Другие инструменты, такие как CruiseControl, применяют другой пользовательский интерфейс, предоставляемый как необязательный элемент. Любой хорошо проработанный пользовательский интерфейс сэкономит ваше время и силы при работе с инструментом.
- **Публикация артефактов**. В конце концов, итог успешного построения — это развертываемый компонент. Если вы используете реальную мощь CI, то результат будет также включать документирование, проверку результатов, анализ их качества и другие показатели. Все инструменты обеспечивают некоторый уровень функций публикации, предоставляя каталог для содержания публикуемых артефактов. Более сложные инструменты автоматически оформляют результаты проверки разработчика и другие отчеты для удобства просмотра.
- **Защита**. И наконец, некоторые инструменты обеспечивают аутентификацию и авторизацию, позволяя определять, кто именно может просматривать результаты и вносить изменения в конфигурацию. Обычно, учитывая корпоративный дух CI, в этом нет необходимости, но если вы работаете несколькими группами или имеете специфические требования безопасности, то это может оказаться важным. Тем не менее помните, что обеспечение защиты увеличивает затраты на поддержку. Каждый раз, когда некто входит в состав группы или покидает ее, вам придется модифицировать базу данных прав доступа инструмента.

Совместимость со средой

Под совместимостью понимается то, насколько хорошо инструмент интегрируется с другими элементами процесса разработки программного обеспечения. При оценке инструментов построения желательно иметь в его составе компилятор для языка, на котором вы работаете. Поддерживает ли он вашу систему с контролем версий? Все это немаловажно. Кроме того, следует учитывать еще и такие вопросы.

- **Поддерживает ли инструмент текущую конфигурацию построения?** Предположим, вы работаете над проектом Java, в котором все еще используется JDK 1.2 и устаревшие элементы (возможно, это правительственный проект). Будет ли инструмент совместим с этим выпуском Java и сможете ли вы настроить JDK, который нужно использовать для компиляции и запуска? Большинство инструментальных средств вполне может быть настроено так, чтобы осуществлять построение для любой платформы, но это все же следует проверить.
- **Требует ли инструмент для запуска установки дополнительного программного обеспечения?** В идеальном случае лучше установить новый инструмент и сразу правильно его настроить. В других случаях вам, возможно, придется установить некоторое дополнительное программное обеспечение, прежде чем начинать его использовать. Например, большинство планировщиков построения Java, которые мы рассмотрим в этом приложении позже, требуют установки Web-сервера с контейнером сервлета. Некоторые инструменты для запуска могут потребовать установки новой среды, например Python или Ruby. Следует также обратить внимание на дополнительные усилия, необходимые для инсталляции и поддержки этого обязательного программного обеспечения. Как правило, их поддержка необременительна, но лучше обойтись без нее.
- **Написан ли инструмент на том же языке, что и проект?** Чем глубже разработчик инструмента будет вынужден вникать в проблемы, с которыми придется столкнуться и вам, тем лучше его инструмент с ними справится. В случае программного обеспечения с открытым исходным кодом вы при необходимости сможете запустить инструмент в отладчике. Кроме того, если вы хорошо разбираетесь в вопросах CI, то сможете дополнить инструмент своими разработками и поспособствовать развитию сообщества его пользователей.

Надежность

Это, в основном, вопрос зрелости инструмента. Если вы не хотите упражняться в системном программировании, то вам необходим инструмент, проверенный временем. Стоит ли говорить, что средство выпуска 3.0, вероятно, будет существенно надежней бета-версии другого инструмента.

Еще одним важнейшим показателем зрелости инструмента является величина сообщества разработчиков и пользователей. Поддержку некоммерческого программного обеспечения осуществляют его пользователи, таким образом, чем больше их сообщество, тем проще получить ответы на вопросы. Проверьте архив списка адресатов поддержки инструмента. Не слишком ли велика активность? Сколько разработчиков внесло свой вклад в проект инструмента с открытым исходным кодом? Насколько велика активность разработчиков в последнее время? Как часто загружают (download) инструмент? Кроме того, если он имеет длительную и легендарную историю, то это хороший признак его популярности.

Долговечность

В то время как надежность инструмента определяется его прошлым и настоящим, долговечность определяет его будущее. Могу побиться об заклад, что ни об одном из описанных далее инструментов даже не вспомнят через 1 000 лет, но при этом вам, вероятно, не нужен инструмент, который продержится до следующего месяца.

Здесь также стоит выяснить популярность у пользователей и величину группы разработчиков. Применяется ли инструмент большим и процветающим сообществом или это очередное "чудесное" решение, имеющее "свой секрет"?

Хотя это и не очевидно, но долговечность выше у инструментов с открытым исходным кодом, поскольку сообщество пользователей постоянно поддерживает их актуальность. Хороший инструмент с уникальными возможностями продолжает использоваться, а инструмент, ничем не выделяющийся, выходит из моды очень быстро. Продолжительность существования коммерческого продукта зависит от его экономической жизнеспособности, а также от производителя. Все мы наблюдали случаи, когда удобный, хорошо проработанный коммерческий продукт превращался в тяжеловесного, непригодного к эксплуатации монстра в результате непрерывного добавления возможностей. Выбор коммерческого инструмента — это вовсе не обязательно плохое решение. Некоторые коммерческие инструменты обладают такими возможностями, которых нет ни у одного предложения с открытым исходным кодом. Только имейте в виду, что ваш сервер CI станет вашим близким другом, и вы захотите поддерживать его как можно дольше, прежде чем распроститься с ним навсегда.

Применимость

И наконец, чем проще инструмент в настройке и применении, тем лучше. Вы можете поэкспериментировать с несколькими инструментами, чтобы оценить это. Как правило, единственное различие в использовании, которое вы можете обнаружить у разных инструментов, заключается в настройке параметров нового проекта, которые приходится задавать в его начале. Инструмент моего выбора — это сервер CruiseControl, и я обычно пишу его файл XML конфигурации вручную (хотя для этого есть специальное приложение GUI). Написание кода XML вручную — это, конечно, менее дружественный подход, чем Web-интерфейс, предоставляемый большинством других средств, однако, на мой взгляд, различие во времени первоначальной настройки намного менее важно по сравнению с преимуществами использования сервера CruiseControl для моих проектов.

Теперь, ознакомившись с различными аспектами, которые следует учитывать при оценке инструментов CI, рассмотрим наиболее важные из них. Итак, начнем с инструментальных средств, доступных в настоящее время.

Инструменты автоматизации построения

Выбор инструмента автоматизации построения довольно прост. Если речь идет о языке Java, то для управления проектом вы, вероятно, используете Ant или Maven, поскольку они предоставляют для этого прекрасные средства. Если вы работаете с платформой .NET, то, скорее всего, применяете NAnt или MSBuild.

В данном разделе приводится краткий обзор автоматизированных инструментов построения. Безусловно, их перечень не является исчерпывающим; например, мы не рассматриваем здесь инструменты построения, связанные с IDE, и автономные инструменты построения, обладающие GUI.

Ant

Дистрибьютор: Apache (`http://ant.apache.org`).
Платформа: Java.
Требования: JDK 1.2 или позднее.

На момент настоящей публикации Ant являлся наиболее широко используемым инструментом построения для Java. Его обширные функциональные возможности перечислены в приложении ранее. Поскольку применение Ant описывалось ранее в этой книге, я просто повторю, что построение Ant определяется с использованием файла конфигурации XML (`build.xml`) и запускается из командной строки или при помощи других инструментов (с которыми он интегрирован), таких как IDE и инструментов планирования построения.

Первоначально Ant был выпущен компанией Apache для собственного использования в 2000 году, но со временем стал одним из наиболее широко применяемых инструментов Java в мире. Он хорошо документирован и весьма надежен. Просто не забывайте, что Ant имеет смысл рассматривать как инструмент построения для проекта Java в первую очередь. Единственной достойной альтернативой Ant является Maven и некоторые рассматриваемые далее коммерческие инструменты, работающие на более высоком уровне, чем Ant, и зачастую "внутренне" использующие функциональные возможности Ant.

Maven 1

Дистрибьютор: Apache (`http://maven.apache.org/maven-1.x/`).
Платформа: Java 2.
Требования: JDK 1.4 или позднее.

Maven от Apache — это инструмент с открытым исходным кодом, работающий на более высоком уровне, чем типичные инструменты построения. На своем Web-сайте Maven описан как "инструмент управления проектом программного обеспечения". После небольшой настройки Maven способен осуществлять построение проекта программного обеспечения, выполнять проверки разработчика, писать множество полезных отчетов о качестве исходного кода, а также создавать Web-сайт, содержащий информацию о перечисленных шагах.

Установка Maven довольно проста. Для платформы Windows предусмотрен инсталлятор; на других платформах достаточно просто распаковать поставляемый комплект, установить переменную среды окружения `MAVEN_HOME` и добавляющей Maven в путь (`path`). Также возможна интеграция со следующими IDE: IntelliJ IDEA, Eclipse, JBuilder и JDEE.

Чтобы настроить проект на использование Maven, сначала следует создать файл `project.xml` в корневом каталоге проекта с его описанием. Его простейший пример приведен в листинге Б.1. Информация в файле `project.xml` определяет то, что известно как *объектная модель проекта* (Project Object Model — POM). POM описывает широкий диапазон информации проекта, от основной (место расположения структуры каталогов проекта) до высокоуровневой информации (информации о подписке, списки адресов пользователей и разработчиков).

Листинг Б.1. Пример простого файла `project.xml`

```
1    <project>
2        <id>helloworld</id>
3        <name>Hello World</name>
4        <version>1.0-SNAPSHOT</version>
5        <organization>
6            <name>Continuous Integration Book</name>
7        </organization>
```

```
8         <description>Our Hello World project</description>
9         <build>
10            <sourceDirectory>src/java</sourceDirectory>
11            <unitTestSourceDirectory>src/test</
↳unitTestSourceDirectory>
12                <unitTest>
13                    <includes>
14                        <include>**/*Test.java</include>
15                    </includes>
16                </unitTest>
17        </build>
18    </project>
```

Одно из главных преимуществ Maven заключается в том, что в отличие от такого инструмента построения, как Ant, который требует явного описания всего необходимого для построения, Maven предоставляет весьма удобные параметры по умолчанию для создаваемого проекта, а также то, какие артефакты должны быть созданы. Это вовсе не означает, что Maven недостает гибкости; вы можете легко перенастроить модель POM, переопределив или дополнив стандартное поведение Maven. Maven обладает дополнениями, которые можно использовать для всех артефактов построения J2EE, вплоть до написания дополнительных отчетов. Вы можете также дополнить Maven, написав собственные дополнения или сценарии.

Еще один интересный момент — это то, как Maven обрабатывает зависимости, включая файлы JAR, обязательные для построения проекта и необходимые Maven для его собственных внутренних функциональных возможностей. Вместо того чтобы включать собственные библиотеки JAR внутрь проекта, вы объявляете их его зависимостями, и Maven выполняет задачу их загрузки из центрального хранилища в кэш на машине, где установлен Maven.

Maven используется при вводе команды **`goal`** в командной строке. Он аналогичен целевой задаче в других инструментах построения. Например, ввод команды **`maven clean`** в командной строке удалит все результаты построения и другие созданные артефакты. Ввод команды **`maven build`** приведет к построению проекта и запуску его комплекта проверки JUnit. Одной из наиболее интересных задач является `site`, которая осуществляет построение проекта, его проверку, написание отчетов и публикацию Web-сайта проекта. Это стандартная страница обобщенного резюме проекта, созданная на основании файла `project.xml`, как показано в листинге Б.1.

Важно понять, что Maven разработан для создания одного артефакта построения на проект, будь то JAR, WAR или EAR. Если ваш проект состоит из нескольких файлов JAR и других файлов, каждый из них потребует собственного отдельного проекта Maven с зависимостями от других проектов, объявляемых по мере необходимости. В Maven 2 намного проще объединить разнообразные артефакты построения в единый проект.

Большинство инструментов планирования построения Java поддерживают интеграцию как с Maven, так и с Ant. Существует также отдельная разновидность проекта Maven под названием Continuum (обсуждается далее), который обеспечивает планирование построения. Если вы решили использовать Maven, то убедитесь, что инструмент планирования построения, который вы выбрали, совместим с ним.

В целом Maven вполне достоин рассмотрения, если вам не нужен абсолютный контроль, предоставляемый низкоуровневыми инструментами построения. Maven предоставляет достаточно много функциональных возможностей при относительно небольшом объеме дополнительных затрат на настройку.

Maven 2

Дистрибьютор: Apache (`http://maven.apache.org`).
Платформа: Java.
Требования: JDK 1.4 или позднее.

Maven 2 продолжает традиции Maven 1 в смысле легкости эксплуатации. Простота в использовании достигается за счет предоставления общей структуры проекта и единообразия системы построения. Кроме того, Maven 2 предоставляет стандартизированную информацию проекта, руководство, полезные советы, а также простую инструкцию по переходу с Maven 1.

По сравнению с предыдущей версией, Maven 2 обладает рядом существенных усовершенствований. Он намного быстрее и меньше по размеру. К другим усовершенствованиям относится улучшенное управление зависимостями (поддержка переходных зависимостей), определение жизненного цикла построения, улучшенная архитектура дополнений и объединенное определение проекта.

Установка и запуск Maven 2 довольно просты, но сначала необходимо загрузить последний инсталлятор, доступный на `http://maven.apache.org/download.html`. Вы можете легко создать каркас проекта с очень простой структурой и минимальным количеством файлов. Достаточно ввести команду **`mvn archetype:create -DgroupId=my.group.id -DartifactId=my-artifact-id`**, и Maven 2 построит проект, совместимый со стандартной структурой каталога. Теперь вы можете добавить собственные классы Java и сформировать проект, введя команду **`mvn clean package`**.

Одна из возможностей, делающая Maven 2 универсальным, — это доступность большого количества дополнений с открытым исходным кодом. В набор базовых дополнений Maven 2 компания Apache включила общие задачи, такие как компиляция и развертывание, упаковка (файлы EJB, JAR, RAR, WAR и EAR), создание отчетов, инструменты и IDE создания проектов. Maven 2 поддерживается также проектом Mojo от Codehaus. Mojo предоставляет множество дополнений, от ассемблера и AspectJ до XML и xdoclet. Использовать дополнения очень просто, следует всего лишь объявить их в файле POM. Инструмент достаточно интеллектуален, чтобы найти дополнение в Интернете и загружать его во временный каталог на локальном диске, настроить его, запустить соответствующую задачу и сообщить о результатах ее выполнения. Весьма полезная возможность, требующая всего четырех строк кода.

Maven 2 обеспечивает также поддержку и для IDE. Компания Codehaus, владелец Mojo, распространяет Mergere для Eclipse и Mevenide для NetBeans. Оба дополнения позволяют открыть файл проекта Maven 2 (POM) внутри IDE и запускать задачи Maven полностью из IDE.

С учетом всех описанных выше преимуществ данного инструмента почему бы не рассмотреть его в качестве системы построения? Дело вкуса. Если вы имеете большой корпоративный проект с многочисленными сценариями Ant, их перенос и смена компоновки проекта может отнять слишком много времени. Maven 2 позволяет вызвать целевые объекты Ant из файла POM, который потенциально может облегчить переход; однако некоторое планирование все же понадобится. С другой стороны, если вы начинаете новый проект, такие возможности, как его стандартизированная компоновка, управление зависимостями, автоматическая документация и доступность высокопроизводительных дополнений стороннего производителя, способно поместить Maven 2 в начало списка кандидатов на систему построения. Большинство серверов CI, включая CruiseControl, обеспечивают поддержку для Maven 2. Рис. Б.1 демонстрирует сайт проекта, созданный Maven.

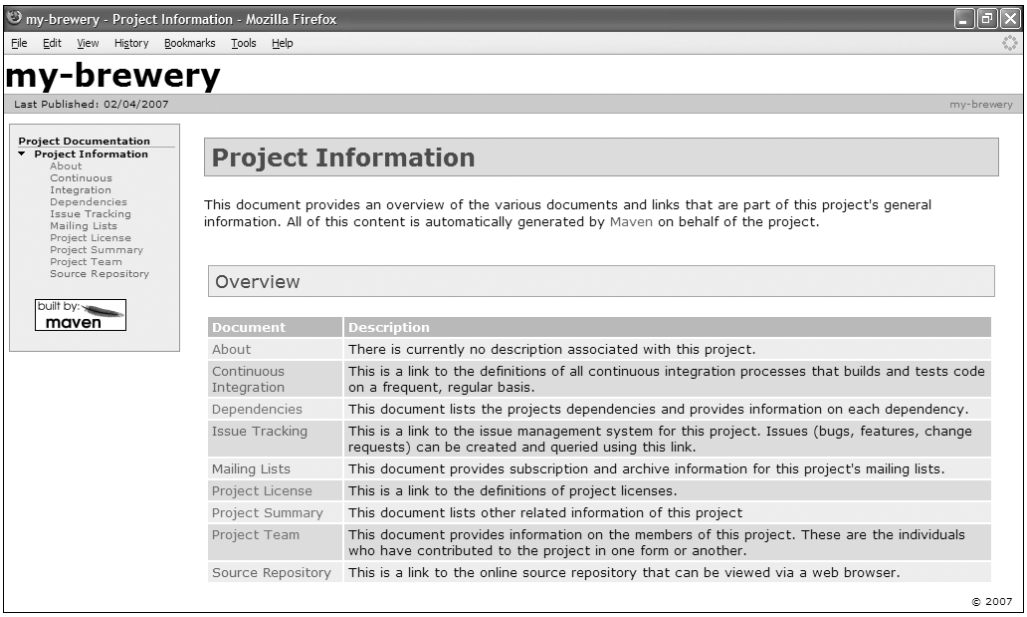


Рис. Б.1. Сайт проекта, созданный Maven

NAnt

Дистрибьютор: SourceForge (`http://nant.sourceforge.net`).
Платформа: Microsoft .NET.
Требования: Microsoft .NET Framework 1.0 и позднее или Mono (профиль 1.0 или 2.0).

NAnt — это автоматизированный инструмент построения с открытым исходным кодом для проектов Microsoft .NET. Как следует из названия, он очень похож на инструмент Ant по настройке и работе. Подобно Ant, NAnt использует файл построения XML, чтобы определить параметры построения проекта. В листинге Б.2 приведен пример типичного файла построения, используемого при компиляции одиночного файла исходного кода C#. Имена файлов построения должны иметь расширения `.build`.

Листинг Б.2. Пример файла построения NAnt

```
<project name="Hello World" default="build" basedir=".">
    <target name="clean">
        <delete file="HelloWorld.exe" failonerror="false" />
    </target>
    <target name="build">
        <csc target="exe" output="HelloWorld.exe">
            <sources>
                <include name="HelloWorld.cs" />
            </sources>
        </csc>
    </target>
</project>
```

NAnt предоставляет функциональные возможности как задачи, которые вызываются из целевых объектов (`target`), определенных в файлах построения. NAnt имеет задачи для компиляции программ, написанных на языках C, C++, C#, J#, Visual Basic.NET и JScript.NET. Другие задачи NAnt обеспечивают функциональные возможности для манипулирования файлами, создания файлов AssemblyInfo, регистрации служб .NET, запуска проверок модуля NUnit и обращения к хранилищу с контролем версий CVS.

Построение запускается из командной строки в результате вызова NAnt и передачи ему в качестве аргумента имени целевого объекта. Например, для запуска целевого объекта `clean`, как в листинге Б.2, вам необходимо ввести в командной строке команду **`nant clean`**. В файлах построения можно также объявить целевой объект, заданный по умолчанию, который запускается, если имя целевого объекта не указано. Строка 1 листинга Б.2 объявляет, что целевым объектом по умолчанию будет `build`.

Инструмент NAnt доступен начиная с 2001 года. Хотя он все еще на фазе бета-выпуска, он широко применяется и очень надежен. Следует заметить, что начиная с Visual Studio 2005 корпорация Microsoft ведет борьбу с ее собственным ориентированным на дескрипторы XML инструментом построения по имени MSBuild. Оба инструмента, NAnt и MSBuild, являются хорошим выбором для автоматизации построения вашего проекта .NET.

Rake

Дистрибьютор: RubyForge (`http://rake.rubyforge.org/`).
Платформа: Ruby и другие платформы разработки.
Требования: Ruby 1.8 или позднее.

Rake — это make для Ruby, однако его уникальность в том, что файлы Rake по существу являются сценариями Ruby, а не XML или чего-то другого. Следовательно, использовать Rake невероятно просто. Очень похожий на Ant для Java, Rake имеет понятие *задач*, которые могут зависеть от других задач; кроме того, Rake поставляется с набором задач, включая проверки разработчика, создание RDocs и множество файловых утилит. Но самое интересное, что Rake — мощнейший язык построения, способный обеспечивать таковое для других языков типа Java.

В листинге Б.3 приведен пример файла Rake, запускающего все проверки модулей, определенные в каталоге `tests/unit/`.

Листинг Б.3. Пример файла Rake, запускающего проверки модуля

```
require "rake/testtask"

task :default => [:unit-test]

Rake::TestTask.new(:unit-test) do | tsk |
 tsk.test_files = "tests/unit/**/*Test.rb"
end
```

Обратите внимание, что во второй строке определена задача по умолчанию (`default`) для проверки модуля (`unit-test`). Это значит, что при вызове Rake через командную строку без аргументов будет запущена задача проверки модуля.

Создание зависимости задачи Rake довольно просто; фактически, вы можете увидеть это в действии на примере листинга Б.3. Задача `default` имеет неявную зависимость от проверки `unit-test`. Внутри определений задачи вы можете также определять зависимости. Например, перед созданием документации исходного кода, вероятно, имеет смысл

запустить все проверки модулей; следовательно, в файл Rake (листинг Б.4) можно добавить задачу RDoc, которая напрямую зависит от задачи проверки модуля.

Листинг Б.4. Пример файла Rake с зависимостями

```
require "rake/testtask"
require "rake/rdoctask"

task :default => [:unit-test]

Rake::TestTask.new(:unit-test) do | tsk |
 tsk.test_files = "tests/unit/**/*Test.rb"
end

Rake::RDocTask.new(:rdoc => [:test]) do | tsk |
  tsk.rdoc_files.include("./src/ruby/*.rb")
end
```

Вполне очевидно, что для приложений, разрабатываемых в среде Ruby, Rake — наилучший выбор. Как уже упоминалось, Rake подходит для построения приложений и в других средах.

Инструменты планирования построения

С учетом разнообразия инструментов планирования построения и их популярности несложно заметить, что концепция CI получила широкое распространение. В данном разделе мы исследуем наиболее популярные из этих инструментов, подходящие для проектов .NET и Java. Как уже упоминалось, мы не будем рассматривать все инструментальные средства, доступные на рынке. Здесь представлены лишь наиболее известные из них (а также некоторые интересные новинки), поскольку категория данных инструментов постоянно пополняется. Эти универсальные средства разработаны так, чтобы выполняться на едином сервере построения и легко обрабатывать большинство проектов, что относится к большей части инструментальных средств, представленных в этом приложении. В данном разделе вы найдете как инструменты с открытым исходным кодом, так и коммерческие средства. Для каждого инструмента указано, является ли он коммерческим или с открытым исходным кодом, затем перечислены предпосылки системы, поддерживаемые инструменты построения и системы с контролем версий.

AnthillPro

Дистрибьютор: Urbancode (www.anthillpro.com/).

Платформа: Java.

Инструменты построения: Ant, GNU Make, Maven, NAnt и командная строка.

Системы с контролем версий: AccuRev, ClearCase, CVS, MKS, Perforce, PVCS, StarTeam, Subversion и Visual SourceSafe.

Требования: JDK 1.4 или позднее.

Компания Urbancode создала Anthill OS в 2001 году как бесплатный инструмент для управления построения. Поскольку данный продукт пользовался успехом, он был выпущен как коммерческий под именем AnthillPro. Функциональные возможности построения AnthillPro основаны на предыдущем Anthill OS, но обеспечивают дополнительные средства, включая более гибкую конфигурацию и новый пользовательский интерфейс. Его основная панель представлена на рис. Б.2. Компания Urbancode предоставляет пробную версию, доступную для загрузки с ее Web-сайта, так что вы можете сами опробовать ее.

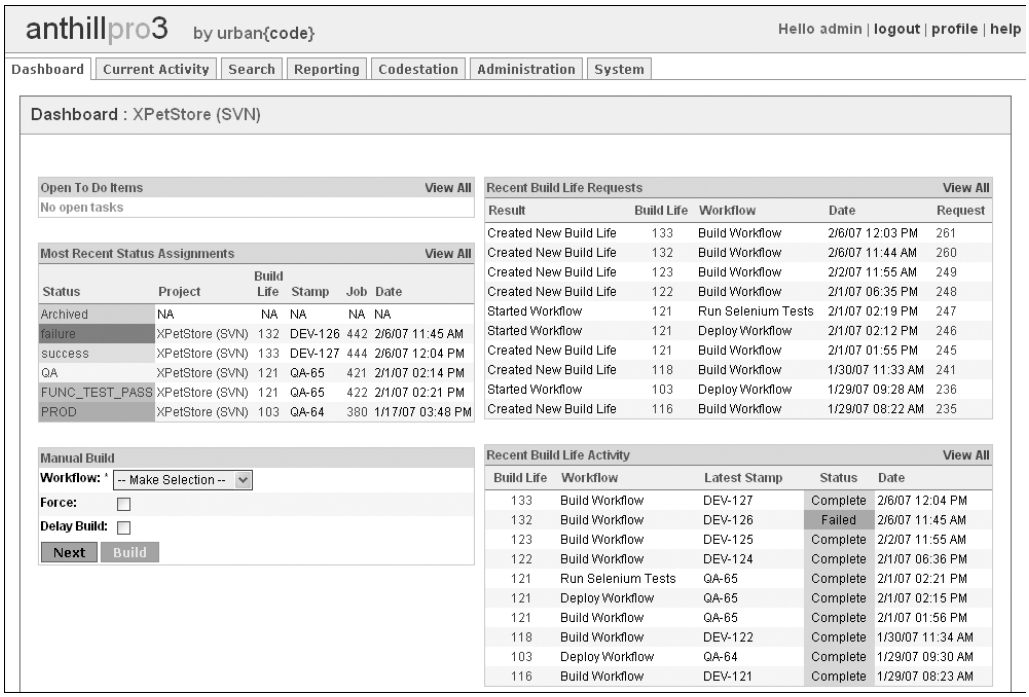


Рис. Б.2. Основная панель AnthillPro

AnthillPro обладает рядом дополнительных возможностей по сравнению с Anthill OS. Во-первых, AnthillPro имеет адаптеры для нескольких дополнительных версий провайдеров управления. Во-вторых, AnthillPro обеспечивает поддержку для Maven и GNU Make, а также возможность интеграции с Ant. Для некоторых наиболее полезным добавлением могут оказаться средства авторизации и аутентификации. Эти новые функциональные возможности позволяют администраторам указать, кому именно разрешено просматривать и редактировать параметры настройки, а также кто и как может обращаться к артефактам построения. Поскольку данный инструмент предоставляет не только исчерпывающие средства для управления построением и CI, он обладает разнообразными возможностями для построения разных типов (отличных от интеграционного построения в течение цикла разработки) и проектирования зависимостей, а также несколькими другими утилитами.

Установка, по сути, состоит в извлечении устанавливаемого файла JAR при помощи командной строки. Инструмент AnthillPro очень гибок в конфигурации, позволяя пользователям настраивать различные профили JVM и инсталлировать Ant, чтобы применять их при построении. Подобно Anthill OS, AnthillPro также управляется по расписанию. Новые расписания можно определить либо как простые интервалы, либо как выражения `cron`.

Настройка AnthillPro может озадачить пользователей-новичков — повышенная гибкость воплощена в обширном массиве параметров, которые могут быть не сразу понятны. Как зачастую бывает с инструментальными средствами, обладающими увеличенным набором функциональных возможностей, это может затруднить настройку, хотя и не настолько, чтобы снизить популярность инструмента.

Создание нового построения в AnthillPro подразумевает несколько шагов. Сначала вы разрабатываете новый проект, в котором указываете хранилище с контролем версий

и применяемую стратегию маркировки. После этого выбираете ветвь хранилища с контролем версий, используемую для построения проекта. Зачастую это основная линия (называемая также магистралью), но данная возможность может также использоваться для поддержки различных конфигураций, например разрабатываемой ветви, ветви выпуска и ветви исправления ошибок. Каждая ветвь позволяет настраивать разные "циклы построения". Каждый цикл построения может иметь собственное расписание, стратегию публикации и процесса построения. Например, вы могли бы настроить почасовую работу Ant по построению в течение дня, с учетом компиляции и проверки, а также выполнение Maven публикации сайта один раз ночью после полной проверки системы.

AnthillPro обеспечивает высокую гибкость для тех, кому необходимо большее количество настроек по сравнению с Anthill OS. Если вы ищете инструмент, способный на большее, чем CI (но все еще поддерживающий возможности CI), это, вероятно, самый подходящий выбор.

Continuum

Дистрибьютор: Apache (`http://maven.apache.org/continuum/`).

Платформа: Java 2.

Инструменты построения: Ant, Maven 1, Maven 2 и Shell.

Системы с контролем версий: Bazaar, CVS, Perforce, StarTeam и Subversion. Частичная поддержка для ClearCase, Visual Source Safe и файловых систем.

Требования: Java JDK 1.4 или позднее.

К преимуществам Continuum относится поддержка большинства ведущих версий инструментов управления на рынке, включая Subversion и CVS, в планах StarTeam, ClearCase и Perforce. Continuum обладает простой Web-ориентированной установкой и пользовательским интерфейсом. Существует также возможность дистанционного управления, осуществляемая при помощи XML-RPC и SOAP. Наряду со множеством других серверов Continuum предоставляет различные механизмы обратной связи, включая электронную почту и мгновенный обмен сообщениями (IRC, Jabber и MSN). Если Continuum не подходит по скорости, имеются и другие Java-ориентированные серверы CI, например CruiseControl, уже обладающий поддержкой для Maven 2. Рис. Б.3 иллюстрирует пример настройки проекта Continuum для Ant.

CruiseControl

Дистрибьютор: ThoughtWorks (`http://cruisecontrol.sourceforge.net`).

Платформа: Java 2.

Инструменты построения: Ant, Maven 1, Maven 2 и NAnt.

Системы с контролем версий: ClearCase, CM Synergy, CVS, MKS, Perforce, PVCS, Snapshot CM, StarTeam, Subversion, Surround SCM и Visual SourceSafe.

Требования: Java JDK 1.3 или позднее.

CruiseControl — продукт с открытым исходным кодом, наиболее широко используемый сервер CI для Java. В отличие от других описанных в этом приложении универсальных инструментов планирования построения Java, которые упакованы в монолитные Web-приложения, CruiseControl упакован в несколько компонентов дополнения типа главной службы CruiseControl, необязательного Web-приложения для создания отчетов и необязательного GUI конфигурации Swing.

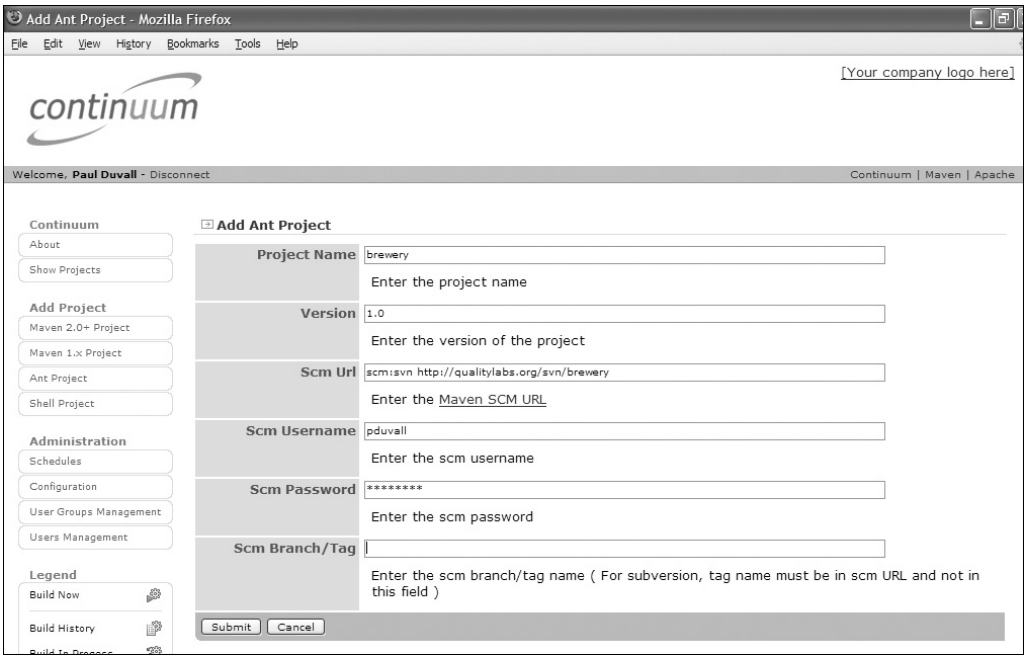


Рис. Б.3. Настройка проекта Continuum для Ant

CruiseControl обычно устанавливают для выполнения в качестве фонового процесса, поддерживающего Web-приложения Java и внешний интерфейс для отчетов. Краткий обзор настройки сервера CruiseControl приведен в главе 1, "Первые шаги". Новичкам исходные настройки не всегда понятны, по крайней мере по сравнению с инструментами, которые поддерживают Web-ориентированный интерфейс конфигурации. Вы, вероятно, согласитесь, что использование GUI настройки Swing поможет ускорить конфигурацию. Даже несмотря на это, уделите время изучению руководств по настройке, доступных на сетевых ресурсах, это окажет существенную помощь. Не забывайте, файл `config.xml` критически важен для правильной настройки сервера CruiseControl.

Кроме эффективного процессора и поддержки для широкого диапазона систем с контролем версий, сервер CruiseControl предоставляет дополнительные средства, которыми не наделен ряд других инструментов. Если вы используете CruiseControl для автоматизации нескольких проектов, вы можете настроить его так, чтобы выполнялось несколько потоков, обеспечивающих параллельное построение. При желании, применяя FTP или SCP (Secure Copy), артефакты построения можно расположить на дистанционных серверах. CruiseControl предоставляет также интерфейс JMX, применяемый для дистанционной настройки или автоматизации самой службы CruiseControl.

С учетом его функциональных возможностей, высокой адаптации и отказоустойчивости, сервер CruiseControl, вероятно, следует рассматривать как один из первых кандидатов на роль сервера CI для проектов Java.

CruiseControl.NET

Дистрибьютор: ThoughtWorks (`http://confluence.public.thoughtworks.org/display/CCNET`).

Платформа: Microsoft .NET.

Инструменты построения: MSBuild, NAnt и Visual Studio .NET.

Системы с контролем версий: ClearCase, CVS, MKS, Perforce, PVCS, SourceGear Vault, StarTeam, Subversion, Synergy и Visual SourceSafe.

Требования: Microsoft .NET Framework версии 1.0, 1.1 или 2.0.

Подобно CruiseControl для Java, сервер CruiseControl.NET — наиболее широко используемый сервер CI для проектов .NET. На мой взгляд, установка и настройка довольно просты. Особенно полезны примеры файлов конфигурации, входящие в состав поставки. Они демонстрируют применение большинства параметров настройки построения и контроля версий. Благодаря тому что я использовал сервер CruiseControl на протяжении некоторого времени, а конфигурация сервера CruiseControl.NET очень похожа на него, я был просто поражен, когда сумел подготовить и запустить сервер CruiseControl.NET буквально через несколько минут после установки.

Сервер CruiseControl.NET пригоден не только для запуска задач NAnt и MSBuild, его можно также использовать для автоматизации простых построений с применением Visual Studio .NET (хотя это потребует установки компонентов Visual Studio на сервере построения). К информации о состоянии построения CruiseControl.NET и артефактам построения можно обращаться при помощи необязательного Web-приложения, установка которого не составляет труда. Рис. Б.4 демонстрирует пример результирующей Web-страницы построения.

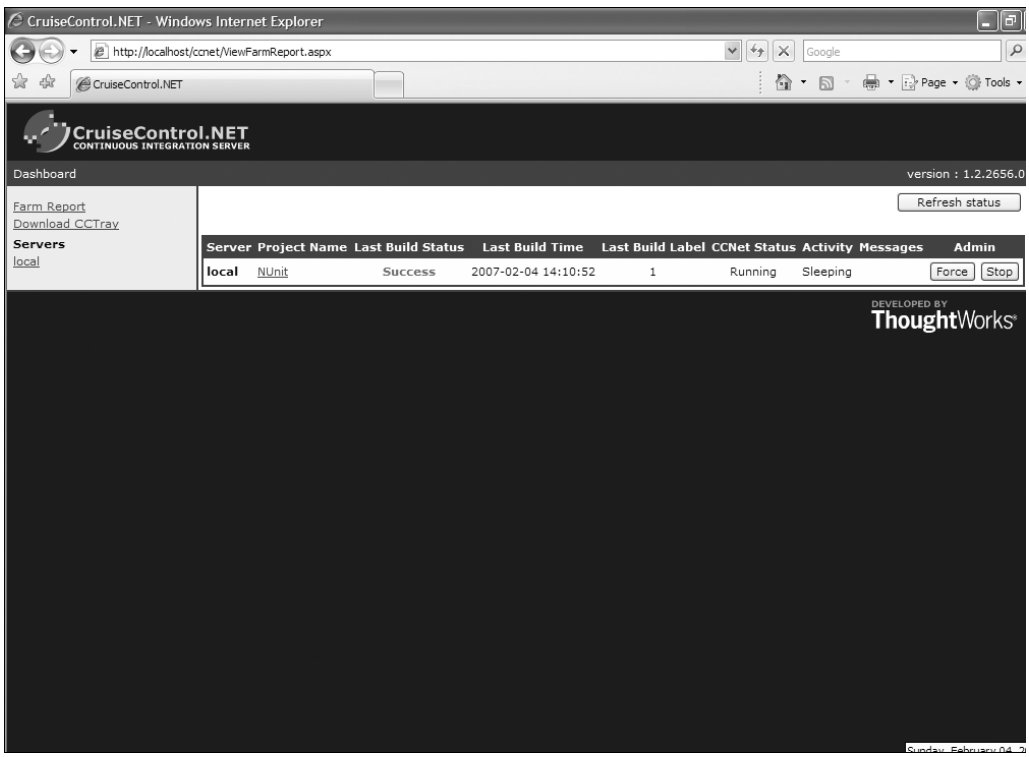


Рис. Б.4. Основная панель CruiseControl.NET

Выпущенный в 2003 году, сервер CruiseControl.NET еще не завоевал такой популярности, как его аналог для Java. Однако несмотря на свою относительную молодость, сервер CruiseControl.NET — весьма надежный инструмент, а его документация и поддержка пользователей превосходны. Если вы планируете реализовать CI для своих проектов .NET, я настоятельно рекомендую применять данный инструмент.

Draco.NET

Дистрибьютор: SourceForge (`http://draconet.sourceforge.net/`).
Платформа: Microsoft .NET.
Инструменты построения: NAnt и Visual Studio .NET.
Системы с контролем версий: CVS, Subversion и Visual SourceSafe.
Требования: Microsoft .NET Framework версии 1.0 или 1.1.

Draco.NET — еще один сервер CI с открытым исходным кодом для платформы .NET. С точки зрения конфигурации и использования он очень напоминает сервер CruiseControl; фактически, авторы Draco.NET черпали вдохновение на домашней странице CruiseControl. Подобно CruiseControl, базовая служба и пользовательские Web-приложения распределены на отдельные компоненты, в данном случае как инсталляторы Windows. Кроме них, Draco.NET обладает клиентским компонентом, который допускает вызов из командной строки сервера построения из дистанционной машины.

Установка проходит очень просто, при этом используется стандартная служба Microsoft Installation. Подобно CruiseControl, построение настраивается с использованием файла XML, в данном случае по имени `Draco.builds.config`. Его пример приведен в листинге Б.5. Документация по настройке Draco.NET содержится в справочном файле, включенном в состав поставки, но она слишком краткая. К счастью, в заданном по умолчанию файле конфигурации Draco.NET содержится множество примеров. Даже в этом случае настройка построения и необязательных пользовательских Web-приложений может оказаться довольно сложным эмпирическим процессом; однако время, потраченное на установку инструмента, не пропадет зря. Draco.NET обычно используют для управления построением NAnt проектов .NET, но если Visual Studio установлен на сервере построения, вы можете также непосредственно вызывать Visual Studio .NET в ходе процесса.

Листинг Б.5. Пример файла `Draco.builds.config`

```
1    <draco xmlns="http://www.chive.com/draco">
2        <pollperiod>600</pollperiod>
3        <quietperiod>60</quietperiod>
4        <timeoutperiod>3600</timeoutperiod>
5        <rootsourcedir>Source</rootsourcedir>
6        <mailserver>mail.5amsolutions.com</mailserver>
7        <fromaddress>draco@5amsolutions.com</fromaddress>
8        <builds>
9            <build>
10           <name>HelloWorldNET</name>
11               <notification>
12                   <email>
13                       <recipient>etavela@5amsolutions.com
↳</recipient>
14                   </email>
15                   <file>
16                       <dir>C:\Draco\Output</dir>
```

```
17                  </file>
18               </notification>
19              <nant>
20                  <buildfile>nant.build</buildfile>
21                  <targets>build</targets>
22              </nant>
23              <cvs>
24                   <cvsroot>:pserver:anonymous@localhost:/cvsrepo
↳</cvsroot>
25                   <module>HelloWorldNET</module>
26               </cvs>
27           </build>
28       </builds>
29   </draco>
```

Хотя Draco.NET не столь широко применяется, как CruiseControl.NET, он имеет немалое количество пользователей. Несмотря на вероятность некоторых сбоев, его установка и настройка приемлемо управляемы. Если вы устанавливаете CI для .NET впервые, вам лучше всего начать с CruiseControl.NET в связи с его большей применимостью и более обширной документацией.

Luntbuild

Дистрибьютор: SourceForge (`http://luntbuild.javaforge.com/`).

Платформа: Java 2.

Инструменты построения: Ant, Maven и командная строка.

Системы с контролем версий: AccuRev, ClearCase, ClearCase UCM, CVS, Perforce, StarTeam, Subversion и Visual SourceSafe.

Требования: JDK 1.3 или позднее, контейнер Java Servlet.

Luntbuild — еще один популярный Web-ориентированный сервер CI для платформы Java с открытым исходным кодом. Как и следовало ожидать, его установка заключается в развертывании файла WAR Luntbuild на существующем процессоре Java Server сервера построения.

Web-ориентированный пользовательский интерфейс может быть несколько запутан и непонятен. Luntbuild обеспечивает больше гибкости, чем другие Web-ориентированные серверы CI, если вы согласны мириться с недостаточной применимостью. На рис. Б.5 приведен пример настройки и установки расписания с использованием Luntbuild.

Luntbuild — относительно недавнее добавление, впервые выпущенное SourceForge в 2004 году, но он надежен и имеет приличное количество пользователей. Его применяемость оставляет желать лучшего. Возможно, следует улучшить его интерфейс. Тем не менее я рекомендовал бы начать с проверенного сервера CruiseControl, если наличие Web-интерфейса для настройки построения не слишком важно для вас.

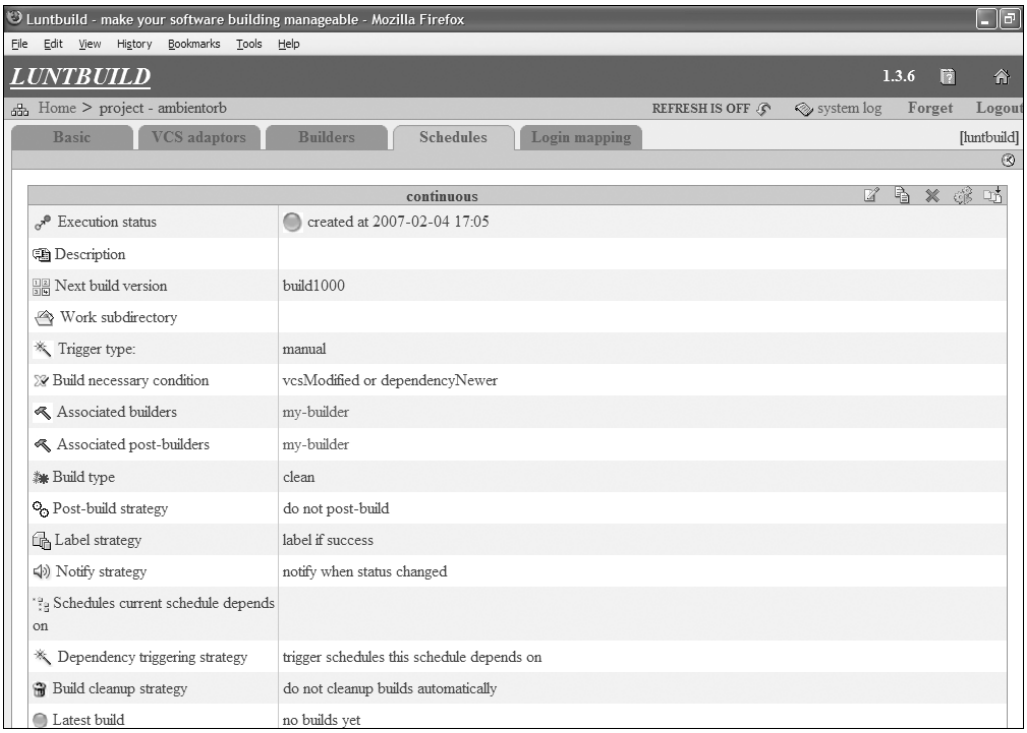


Рис. Б.5. Luntbuild

Заключение

CI становится господствующей тенденцией, имеет инструментальные средства и сообщество пользователей, доказывающих это. Теперь, когда вы готовы присоединиться к тем, кто извлекает выгоду из подходов CI, вы можете выбирать инструменты, обеспечивающие наилучшее соответствие задачам вашего проекта и вашей группы. Хотя мы постарались предоставить вам побольше информации, чтобы облегчить принятие решения, вы должны использовать это приложение лишь как отправную точку в своих исследованиях. Постарайтесь изучить богато представленную в Сети информацию об этих инструментах, их документацию, страницы популярных вопросов и списки адресатов. Вооружившись подобной информацией, вы вполне сможете создать продуктивную реализацию CI.

Библиография

1. Скотт В. Эмблер (Scott Ambler) и Прамодкумар Дж. Садаладж (Pramod Sadalage). *Рефакторинг баз данных: эволюционное проектирование*. ИД: "Вильямс", 2007.
2. Антониол Г., Пента М. Д., Мерло Е. и Виллано У. "Analyzing cloning evolution in the Linux kernel." *Journal of Information and Software Technology*, 44(13):755–765, 2002.
3. Кент Бек (Kent Beck) и Синтия Андрес (Cynthia Andres). *Extreme Programming Explained, Second Edition*. Бостон: Addison-Wesley, 2005.
4. Стивен П. Беркзук (Stephen P. Berczuk) и Бред Апплетон (Brad Appleton). *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Бостон: Addison-Wesley, 2003.
5. Гради Буч (Grady Booch). *Object Solutions: Managing the Object-Oriented Project*. Парк Менло: Pearson Education, 1996.
6. Кусумано Майкл А. "Software Development Worldwide: The State of the Practice" (а также Мак-Кормак Алан, Кемерер Крис и Крендалл Билл), *IEEE Software*, ноябрь–декабрь 2003. Т. 20. Номер. 6. С. 28–34. `www.pitt.edu/~ckemerer/CK%20research%20papers/SwDevelopmentWorldwide_CusumanoMacCormackKemerer03.pdf`.
7. Кусумано Майкл А. и Селби Ричард У. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. Нью-Йорк: Free Press, 1995.
8. Дюваль Поль. *Automation for the People: Choosing a Continuous Integration Server*. `http://www-128.ibm.com/developerworks/java/library/j-ap09056/`.
9. Дюваль Поль. "Automation for the People: Continuous Inspection." `http://www-128.ibm.com/developerworks/java/library/j-ap08016/`.
10. Дюваль Поль. "Automation for the People: Remove the Smell from Your Build Scripts." `http://www-128.ibm.com/developerworks/java/library/j-ap10106/`.
11. Фаулер Мартин. "Continuous Integration." Доступно в сети по адресу `www.martinfowler.com/articles/continuousIntegration.html`.
12. Фаулер Мартин, Бек Кент, Брант Джон, Опдайк Уильям и Роберс Дон. *Refactoring: Improving the Design of Existing Code*. Реадин: Addison-Wesley, 1999.
13. Фаулер Мартин и Дж. Садаладж Прамодкумар. "Evolutionary Database Design." Доступно в Сети по адресу `www.martinfowler.com/articles/evodb.html`.
14. Хант Эндрю и Томас Дэвид. *The Pragmatic Programmer: From Journeyman to Master*. Бостон: Addison-Wesley, 2000.
15. Камия Т., Кусумото С. и Иноуэ К. "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.
16. Мак-Коннелл Стив. *Software Project Survival Guide*. Редмонд: Microsoft Press, 1998.
17. О'Рейли Тим (Tim O'Reilly). "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software." `www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

18. Сиерра Кэйси. "Why 'duh'... isn't." `http://headrush.typepad.com/creating_passionate_users/2006/09/why_duh_isnt.html`.

19. Тумим Майкл, Бегель Эндрю и Грехэм Сьюзен Л. "Managing Duplicated Code with Linked Editing." `http://harmonia.cs.berke-ley.edu/papers/toomim-linked-editing.pdf`.

20. Ван-Дорен Эдмонд. "Cyclomatic Complexity." `www.sei.cmu.edu/str/descriptions/cyclomatic.html`.

21. Веннерс Билл. "Refactoring with Martin Fowler: A Conversation with Martin Fowler, Part I." `www.artima.com/intv/refactor.html`.

22. Вейк Уильям К. "Java Coding Conventions on One Page." `www.xp123.com/xplor/xp0002f/codingstd.gif`.

23. Уотсон Артур Х. и Мак-Каб Томас Дж. "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric." `http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm`.

24. Вилкокс Глен. "Managing Your Dependencies with JDepend."

Предметный указатель

Научно-популярное издание

Поль М. Дюваль, Стивен М. Матиас III, Эндрю Гловер

Непрерывная интеграция:
улучшение качества программного обеспечения и снижение риска

Литературный редактор *Т.Г. Сковородникова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

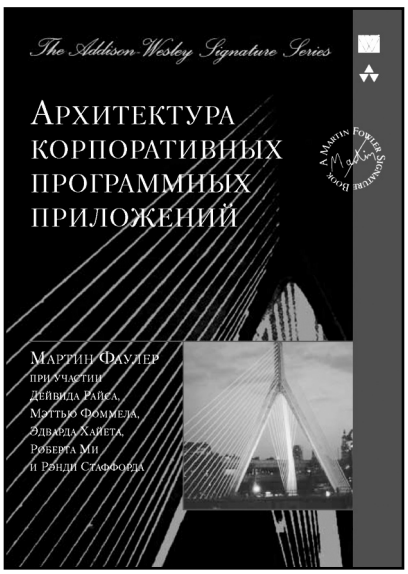
ООО "И.Д. ВИЛЬЯМС"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 31.03.2008. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 19,35. Уч.-изд. л. 15,3.
Тираж 2000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

АРХИТЕКТУРА КОРПОРАТИВНЫХ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

ISBN 978-5-8459-0579-6

в продаже

ПРИМЕНЕНИЕ DDD И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

ПРОБЛЕМНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ С ПРИМЕРАМИ НА C# И .NET

Джимми Нильссон

www.williamspublishing.com

Эта книга о разработке корпоративных программных приложений в среде .NET с применением шаблонов проектирования. В ней описаны: проблемно-ориентированные методы проектирования (DDD, или Domain Driven Design), разработка посредством тестирования (TDD, или Test-Driven Development), объектно-реляционное преобразование, т.е. методы, которые многие относят к ключевым технологиям разработки программного обеспечения. Хотя большинство примеров кода представлено на языке C#, материал книги может оказаться полезным и для тех, кто работает на платформе Java. Книга адресована опытным разработчикам архитектуры и прикладного программного обеспечения уровня предприятий, в том числе и в среде .NET.

ISBN 978-5-8459-1296-1 **в продаже**