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.

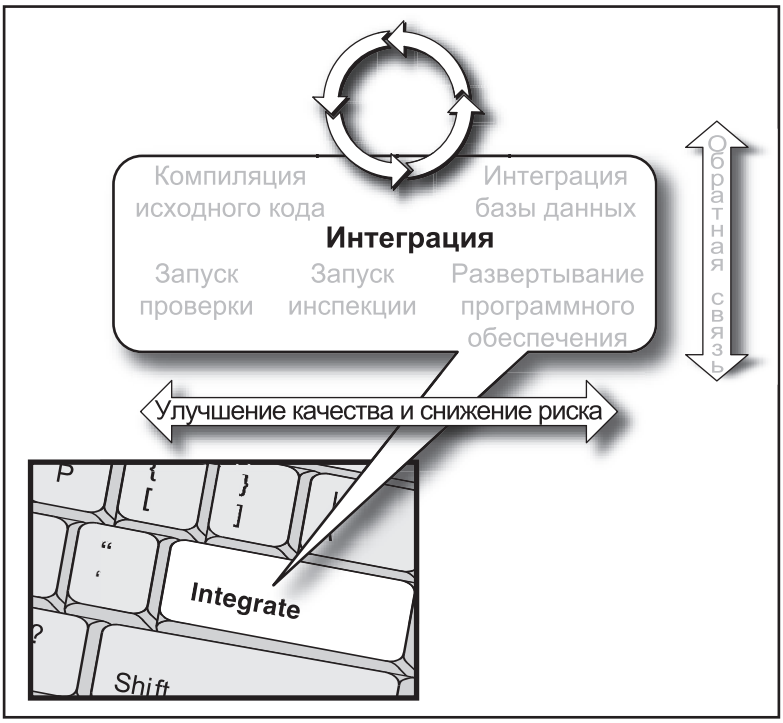

Рис. 3.1. CI может улучшить качество программного обеспечения и снизить риск

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?