Глава 7

Стадия фиксации

Введение

Стадия фиксации начинается с изменения состояния проекта, которое фиксируется в системе управления версиями, а заканчивается отчетом о неудаче или, если завершение стадии успешное, созданием коллекции двоичных артефактов и пригодных к развертыванию сборок, используемых на последующих стадиях тестирования и поставки релиза. Кроме того, на стадии фиксации создаются отчеты о новом состоянии приложения. В идеале стадия фиксации должна выполняться не более пяти минут. Если она выполняется больше десяти минут, значит, нужно приложить усилия к уменьшению этого времени.

Стадия фиксации является точкой входа в конвейер развертывания в нескольких смыслах. Это не только точка, в которой создается и начинает свой путь релиз-кандидат, но и точка, с которой команда разработки начинает реализацию конвейера развертывания. Когда команда реализует процедуру непрерывной интеграции, одновременно она создает стадию фиксации.

Создание стадии фиксации — жизненно важный первый шаг. Она обеспечивает минимизацию времени, затрачиваемого на интеграцию на уровне кодов. Кроме того, она поощряет правильные методики программирования и радикально улучшает качество кода и скорость процесса поставки.

В предыдущих главах мы уже кратко описали стадию фиксации. В данной главе мы расширяем описание, включив подробности создания эффективной стадии фиксации наряду с эффективными тестами фиксации. Эти вопросы интересуют главным образом разработчиков, заинтересованных в обратной связи на стадии фиксации. Структура стадии фиксации показана на рис. 7.1

Для освежения материала в памяти кратко повторим, как она работает. Кто-то из разработчиков регистрирует изменение на магистрали в системе управления версиями. Сервер непрерывной интеграции обнаруживает изменение, находит исходный код и выполняет ряд задач, включая следующие.

- Компиляция (при необходимости) и выполнение тестов фиксации для интегрированного исходного кода.
- Создание двоичных кодов, которые могут быть развернуты в любой среде (если применяется компилируемый язык, этот этап включает компиляцию и сборку приложения).
- Анализ кода, необходимый для поддержания кодовой базы в удовлетворительном состоянии.
- Создание других артефактов (таких, как тестовые данные или реплики баз данных), используемых дальше в конвейере развертывания.

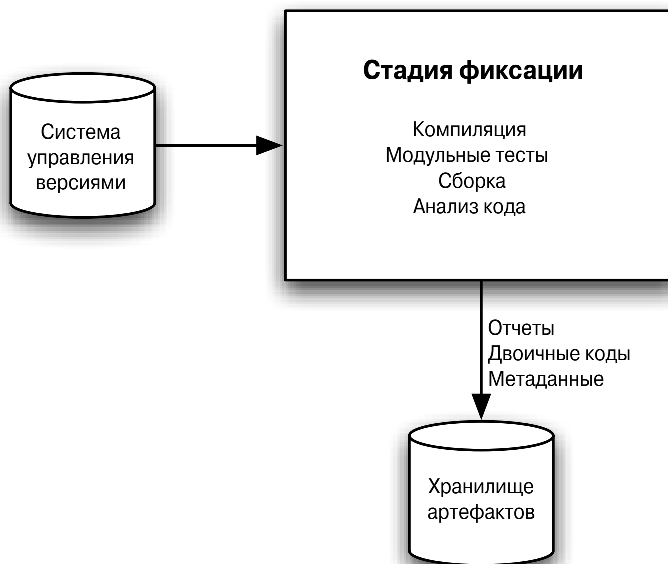

Рис. 7.1. Стадия фиксации

Эти задачи управляются сценариями сборки, запускаемыми сервером непрерывной интеграции. Более подробно сценарии сборки рассматриваются в главе 6. Двоичные коды (в случае успеха стадии фиксации) и отчеты сохраняются в центральном хранилище артефактов для использования командой поставки на более поздних стадиях конвейера развертывания.

Для разработчиков стадия фиксации — наиболее важный цикл обратной связи в процессе разработки. Она предоставляет обратную связь для наиболее распространенных ошибок, вводимых разработчиками в систему. Результат стадии фиксации — важное событие в жизненном цикле каждого релиз-кандидата. Успех на этой стадии — единственный способ входа в конвейер развертывания и, следовательно, инициации процесса поставки приложения.

Принципы и методики стадии фиксации

Одна из целей конвейера развертывания — устранение сборок, непригодных к продвижению в рабочую среду, а цель стадии фиксации — отклонение нежелательных вариантов, перед тем как они могут создать проблемы на более поздних стадиях конвейера. Таким образом, стадия фиксации должна либо создать артефакты, пригодные для развертывания, либо быстро известить команду о неудаче и ее причинах.

Ниже приведен ряд важных методик, используемых на стадии фиксации для обеспечения ее эффективности.

Создайте быструю и надежную обратную связь

Неудача теста фиксации чаще всего обусловлен одной из следующих трех причин: синтаксическая ошибка в коде, перехваченная компилятором; семантическая ошибка

в приложении, перехваченная одним из тестов фиксации; проблема с конфигурацией приложения или среды (включая операционную систему). Какова бы ни была причина неудачи, стадия фиксации должна известить о ней пользователя немедленно по завершении тестов фиксации. Кроме того, стадия фиксации должна сгенерировать краткий отчет о причинах неудачи, например создать список неуспешных тестов и ошибок компиляции. Разработчик должен иметь доступ к консольному выводу стадии фиксации, который для удобства может быть разбит на несколько окон.

Ошибку легче устранить, если она обнаружена рано, близко к точке процесса поставки, в которой она введена. Это обусловлено тем, что в это время разработчик еще хорошо помнит, что и зачем он делал, и понимает ситуацию в системе. Кроме того, на этапе возникновения ошибки механика ее обнаружения проще. Если разработчик внесет изменение, приводящее к неудаче теста, и причина неудачи не очевидна, наиболее естественная процедура поиска источника ошибки состоит в просмотре изменений, внесенных в последнее время, чтобы сузить диапазон поиска.

Если разработчик придерживается наших советов и часто регистрирует изменения, диапазон влияния каждого изменения будет небольшим. Конвейер развертывания должен быстро обнаружить неудачу, в идеале — на стадии фиксации. В таком случае диапазон влияния каждого изменения ограничен тем, что данный разработчик делал лично. Это означает, что устранить проблему, обнаруженную на стадии фиксации, значительно легче, чем идентифицировать ее на более поздних стадиях путем тестирования многих изменений, внесенных другими разработчиками.

Таким образом, чтобы конвейер развертывания был эффективным, нужно перехватывать ошибки как можно раньше. В большинстве наших проектов мы фактически начинаем этот процесс еще до стадии фиксации, применяя современные среды разработки, уделяя много внимания всем предупреждениям этапа компиляции и устраняя синтаксические ошибки, как только они отмечаются средой разработки. Многие современные серверы непрерывной интеграции предоставляют функцию *предварительно протестированной фиксации* (pretested commit), которая проверяет изменения перед регистрацией. Если ваш сервер не предоставляет эту функцию, можете компилировать и выполнять тесты фиксации локально перед стадией фиксации.

Стадия фиксации — первый формальный этап, фокусирующий внимание на параметрах качества, находящихся вне поля зрения индивидуального разработчика. Первое, что происходит на стадии фиксации, — интеграция изменения на магистраль (главную ветвь) версий. При этом выполняется автоматическая проверка процесса выполнения интегрированного приложения. Если вы придерживаетесь концепции раннего обнаружения ошибок, ваша система должна генерировать сигнал неудачи как можно быстрее, и стадия фиксации должна перехватывать как можно больше ошибок, вносимых разработчиками в приложение.

Важно отметить, что при внедрении системы непрерывной интеграции наиболее распространенная ошибка заключается в слишком буквальном понимании концепции “как можно более быстрой генерации сигнала неудачи” и завершении сборки, как только обнаружена ошибка. Это приводит к слишком быстрой оптимизации. В общем случае рекомендуется разделить стадию фиксации на ряд задач (конкретный набор задач зависит от проекта), таких как компиляция, выполнение тестов и т.д. Стадию фиксации следует останавливать, только если ошибка не позволяет выполнить следующие задачи, например ошибка компиляции. В противном случае лучше выполнить стадию фиксации до конца и сгенерировать итоговый отчет обо всех ошибках и неудачах, чтобы можно было исправить все сразу.

Что должно завершать стадию фиксации

Традиционно стадия фиксации разрабатывается таким образом, что она завершается при возникновении одной из следующих ситуаций: неудача компиляции, неудача теста, проблемы со средой. В других случаях стадия фиксации продолжается и генерирует отчет о результате. А что если тесты пройдены только потому, что они не полностью покрывают код? А если качество кода низкое? Или компиляция завершена успешно, но сгенерированы сотни предупреждений? Когда же нужно считать результат удовлетворительным? Незрелая, несовершенная стадия фиксации часто генерирует фальшивый положительный результат, предполагающий, что качество приложения удовлетворительное, хотя на самом деле оно низкое.

Существует несколько спорный подход, заключающийся в отказе от дихотомического представления результата (либо успех, либо неудача). Многие считают его слишком ограниченным. В принципе, можно запрограммировать предоставление более полной информации, например генерацию графа, представляющего покрытие кодов тестами, вычисление других метрик и т.п. Результирующую информацию можно обобщить, задав ряд пороговых значений метрик и представив результат в виде светофора (красный, желтый, зеленый) или стрелочного прибора. Например, можно считать стадию фиксации неуспешной (красный цвет), если покрытие кода меньше 60%, удовлетворительной (желтый цвет), если код покрывается на 60–80%, и успешной (зеленый цвет), если покрытие более 80%.

В реальной жизни мы не встречали реализаций столь изощренных подходов. Тем не менее мы сами писали сценарии стадии фиксации, которые генерируют сигнал неудачи, когда количество предупреждений превышает заданную величину или не снижается по сравнению с предыдущей фиксацией (принцип, который мы назвали “храповик”; см. главу 3). Вы тоже можете запрограммировать генерацию сообщения, когда, например, количество дубликатов кода превышает заданный порог, или при других нарушениях качества кода.

Однако не забывайте, что, если стадия фиксации сообщила о неудаче, команда поставки должна немедленно оставить другие дела и устранить ошибку. Поэтому программируйте генерацию сигнала неудачи только при наличии достаточно веских причин. В противном случае команда перестанет серьезно воспринимать сообщения о неудачах, и система непрерывной интеграции будет разрушена. Непрерывно анализируйте качество приложения и внедряйте в стадию фиксации метрики, вынуждающие поддерживать качество на удовлетворительном уровне.

Непрерывно улучшайте стадию фиксации

Стадия фиксации содержит сценарии сборки, сценарии выполнения модульных тестов, инструменты анализа и т.п. Ко всем этим сценариям нужно относиться не менее “уважительно”, чем к коду приложения. Как и в случае других программных систем, если сценарии сборки разработаны и поддерживаются небрежно, усилия, затрачиваемые на работу с ними, будут возрастать экспоненциально. Плохая система сборки не только отвлекает внимание и время разработчиков от задачи создания деловой логики приложения, но и затрудняет реализацию деловой логики. Мы знакомы с несколькими проектами, в которых создание эффективной деловой логики было затруднено и практически остановилось из-за проблем сборок.

Вы должны непрерывно прилагать усилия, направленные на улучшение качества, структуры и производительности сценариев стадии фиксации. Эффективная, быстрая и

надежная стадия фиксации — ключ к высокой производительности команды разработки, поэтому затраты времени и внимания на ее улучшение почти всегда быстро окупаются. Поддержка скорости и надежности сценариев требует творческого подхода, в частности, правильного выбора тестов и совершенствования их структуры. Сценарии, рассматриваемые как второстепенные задачи, быстро становятся недоступными для понимания и непригодными для поддержки. Например, в одном из наших проектов мы унаследовали сценарий Ant, состоящий из 10 000 строк кода на XML. Естественно, поддерживать такой код было очень тяжело, для этого приходилось содержать целую команду квалифицированных разработчиков, что, естественно, является недопустимой растратой ресурсов.

Убедитесь в том, что сценарии обладают модульной структурой, как описано в главе 6. Структурируйте сценарии таким образом, чтобы общие задачи, часто используемые, но редко изменяемые, были отделены от часто изменяемых задач, таких как добавление новых модулей в кодовую базу. Разместите коды, выполняемые на разных стадиях конвейера развертывания, в разных сценариях. Что самое важное, не создавайте сценарии, специфичные для сред. Для этого отделяйте конфигурации, специфичные для сред, от сценариев сборки.

Передайте полномочия разработчикам

В некоторых организациях есть команды ИТ-специалистов, являющихся экспертами в создании эффективных модульных сценариев сборки приложений и управления средами, в которых они выполняются. Нам самим неоднократно приходилось бывать в этой роли. Тем не менее мы считаем недопустимой ситуацию, в которой поддерживать систему непрерывной интеграции могут только эти эксперты.

Для проекта жизненно важно, чтобы команда поставки чувствовала себя “хозяином” стадии фиксации (как и других стадий конвейера развертывания). Команда должна быть тесно связана со всеми аспектами работы сценариев. Если между разработчиками и сценариями есть любые барьеры, затрудняющие быстрое и эффективное изменение сценариев разработчиками, их производительность будет существенно ухудшена, и в ближайшем будущем обязательно появятся серьезные проблемы.

Любые изменения, как рутинные, так и непредвиденные, такие как добавление новых библиотек или конфигурационных файлов, должны выполняться совместно разработчиками и администраторами при возникновении необходимости в изменениях. Эта работа не должна выполняться экспертами по сценариям сборки (за исключением ранней стадии работы над проектом, когда нужно создать фундамент и шаблоны сценариев).

Без хороших экспертов не обойтись, и нельзя недооценивать их, однако их задача должна быть ограничена созданием хороших структур и шаблонов, выбором технологий и передачей своих знаний команде поставки. Если эти базовые правила установлены, необходимость в экспертах должна появляться, только когда нужно существенно изменить структуру, а в рутинной, повседневной поддержке сценариев команда должна обходиться без них.

В больших проектах для эксперта по средам и сборкам практически всегда есть достаточно работы, чтобы нанять его на полный рабочий день, но наш опыт свидетельствует в пользу того, что знания эксперта используются наиболее эффективно, когда он привлекается лишь время от времени для решения неожиданных сложных проблем, а также для передачи своих знаний команде поставки.

Разработчики и администраторы должны на постоянной основе заниматься поддержкой сценариев сборки и отвечать за их правильное функционирование.

В очень больших командах назначайте администратора сборок

В небольшой, расположенной в одном месте, команде от двадцати до тридцати человек прекрасно срабатывает самоорганизация. Когда сборка испорчена, команда легко находит ответственного за это и, при необходимости, помогает ему решить проблему.

В больших или распределенных командах сделать это намного тяжелее. В таком случае полезно назначить кого-либо администратором сборок. В его обязанности входят контроль над сборками, управление поддержкой сценариев и, что самое важное, поддержка дисциплины сборок. Увидев разрушенную сборку, он мягко (или не очень мягко, если процесс затягивается) напоминает виновнику о его обязанности как можно быстрее исправить сборку или отменить изменение.

Роль администратора сборок полезна также в команде, начавшей заниматься непрерывной интеграцией недавно. В такой команде дисциплина сборок еще слабая, и напоминания необходимы для эффективной работы конвейера.

Однако роль администратора сборок не должна быть постоянной. Члены команды должны получать ее по очереди, приблизительно на неделю. Опыт администрирования будет полезен каждому члену команды. В любом случае людей, желающих получить роль администратора сборок навсегда, не так уж много.

Результаты стадии фиксации

Стадия фиксации, как и каждая стадия конвейера развертывания, имеет вход и выход. На ее вход поступают исходные коды, а на выходе имеем двоичные коды и отчеты. Сгенерированные отчеты содержат результаты тестирования, необходимые для выяснения причин неудачи тестов. Если тесты успешные, отчеты полезны для анализа кодовой базы. В таком случае отчеты могут содержать информацию о покрытии кодов тестами, цикломатической сложности, связанности модулей, а также любые другие метрики, помогающие оценить качество кодовой базы. Двоичные коды, сгенерированные на стадии фиксации, без изменений применяются во всем конвейере и, возможно, войдут в релиз.

Хранилище артефактов

Результаты стадии фиксации — отчеты и двоичные коды — должны где-то храниться для повторного использования на более поздних стадиях конвейера развертывания и для обеспечения доступа к ним членов команды. Наиболее очевидное (на первый взгляд) место для их хранения — система управления версиями. Однако есть ряд веских причин (они перечислены ниже) отказаться от такого решения (кроме нескольких случайных причин, таких как опасность переполнения диска или того, что некоторые системы управления версиями не поддерживают хранение результатов фиксации).

- В отличие от системы управления версиями, в хранилище артефактов нужно записывать только некоторые версии. Если релиз-кандидат терпит неудачу на некоторой стадии конвейера развертывания, он нас больше не интересует. Поэтому можно, при желании, удалить отчеты и двоичные коды из хранилища артефактов.
- Важно иметь возможность отследить историю кода в обратном порядке: от релиза до регистрации изменения в системе управления версиями. Для этого экземпляр конвейера развертывания должен быть согласован с изменениями в системе управления версиями. Регистрация чего-либо в системе управления исходными

кодами как части конвейера развертывания существенно усложняет отслеживание, потому что накапливает другие изменения, связанные с конвейером.

- Одно из требований к стратегии управления конфигурациями заключается в том, что процесс создания двоичных кодов должен быть повторяемым. Это означает, что, если удалить двоичные коды и заново запустить стадию фиксации того же изменения, которое инициировало создание двоичного кода, стадия фиксации должна сгенерировать точно такой же (до последнего бита) двоичный код. В мире управления конфигурациями двоичные коды — “пассажиры второго класса”. С ними можно особенно не церемониться, однако всегда нужно иметь их хеши в постоянном хранилище для проверки вновь создаваемых копий и аудита процесса в обратном порядке: от релиза до стадии фиксации.

Наиболее современные серверы непрерывной интеграции предоставляют хранилище артефактов и наборы конфигурационных параметров, позволяющие периодически очищать хранилище от ненужных артефактов. Обычно они предоставляют декларативные средства задания того, какие артефакты нужно сохранить после выполнения задачи, и содержат веб-интерфейс, позволяющий команде получать отчеты и двоичные коды. Альтернативный подход состоит в использовании выделенных хранилищ артефактов (таких, как Nexus или менеджеры хранилищ в стиле Maven) для обработки двоичных кодов (для хранения отчетов они обычно не предназначены). Менеджеры хранилищ существенно облегчают доступ к двоичным кодам с локальных компьютеров разработки без обращения к серверу непрерывной интеграции.

Создание собственного хранилища артефактов

При желании несложно создать собственное хранилище артефактов. Принципы его создания подробно рассматриваются в главе 13.

На рис. 7.2 показана диаграмма использования типичного хранилища артефактов. В нем хранятся двоичные коды, отчеты и метаданные каждого релиз-кандидата.

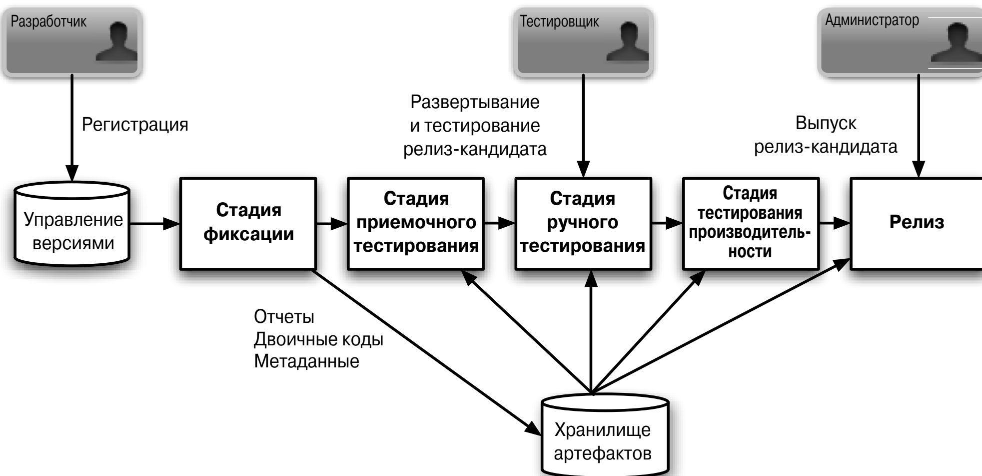

Рис. 7.2. Использование хранилища артефактов

Ниже приведено описание каждого этапа счастливого маршрута релиз-кандидата, успешно продвигаемого в рабочую среду.

1. Разработчик регистрирует изменение.
2. Сервер непрерывной интеграции выполняет стадию фиксации.
3. При успешном завершении двоичные коды, отчеты и метаданные записываются в хранилище артефактов.
4. Сервер непрерывной интеграции извлекает двоичные коды, созданные на стадии фиксации, и развертывает их в тестовой среде, близкой к рабочей.
5. Сервер непрерывной интеграции выполняет приемочные тесты, повторно используя двоичные коды, созданные на стадии фиксации.
6. При успешном завершении релиз-кандидат отмечается как прошедший приемочные тесты.
7. Тестировщики могут получить список всех сборок, прошедших приемочные тесты, и, щелкнув на кнопке, запустить автоматический процесс их развертывания в среде ручного тестирования.
8. Тестировщики выполняют ручное тестирование.
9. При успешном завершении ручного тестирования тестировщики обновляют статус релиз-кандидата, отмечая его как прошедший ручное тестирование.
10. Сервер непрерывной интеграции извлекает из хранилища артефактов последний релиз-кандидат, прошедший приемочное тестирование (или, в зависимости от конфигурации конвейера развертывания, ручное тестирование), и развертывает приложение в рабочей тестовой среде.
11. Релиз-кандидат проходит тесты производительности.
12. При их успешном завершении релиз-кандидат отмечается как успешно прошедший тесты производительности.
13. Данный шаблон повторяется на всех необходимых стадиях конвейера развертывания.
14. Когда релиз-кандидат прошел все стадии, он получает статус готового к выпуску. После этого уполномоченный сотрудник (обычно совместно с администраторами) выполняет поставку релиза.
15. По завершении процедуры поставки релиз-кандидат получает статус релиза.

Примечание

Для простоты маршрут релиз-кандидата описан как последовательный процесс. На ранних стадиях это справедливо, поскольку они выполняются одна за другой. Однако, в зависимости от проекта, после стадии приемочного тестирования часто имеет смысл нарушить последовательность этапов или распараллелить процесс. Например, ручное тестирование и тестирование производительности могут быть запущены одновременно при успешном завершении приемочных тестов. Альтернативный вариант: команда тестирования может выбрать для развертывания в своих средах разные релиз-кандидаты.

Принципы и методики создания набора тестов фиксации

При разработке тестов фиксации необходимо придерживаться ряда принципов. Большинство тестов фиксации являются модульными тестами. На них мы и сосредоточим внимание в данном разделе. Наиболее важное свойство модульных тестов состоит в том, что они должны выполняться очень быстро. Если модульный тест выполняется недостаточно быстро, от него следует отказаться. Второе важное свойство модульных тестов — покрытие кодовой базы. Хорошим считается покрытие 80%. Оно обеспечивает достаточную степень уверенности в том, что, если код прошел модульные тесты, приложение работоспособно. Конечно, каждый модульный тест проверяет только небольшую часть приложения, причем даже не запуская его на выполнение. Поэтому набор модульных тестов не может обеспечить полной уверенности в работоспособности приложения. Впрочем, это от них и не требуется, для этого предназначены другие стадии конвейера развертывания.

Майк Кон [11] дал хорошее визуальное представление того, как должны быть структурированы автоматические тесты (рис. 7.3). Все тесты представлены в виде пирамиды. Подавляющее большинство тестов являются модульными. И хотя их много, они выполняются не дольше нескольких минут. И наоборот, приемочных тестов немного, но они выполняются долго, потому что для них нужно запустить систему. На рис. 7.3 приемочные тесты разбиты на тесты служб и пользовательского интерфейса. Для обеспечения работоспособности приложения и соблюдения правил деловой логики важны все уровни тестирования. Показанная на рис. 7.3 пирамида тестов покрывает левые квадранты диаграммы тестов (см. рис. 4.1).

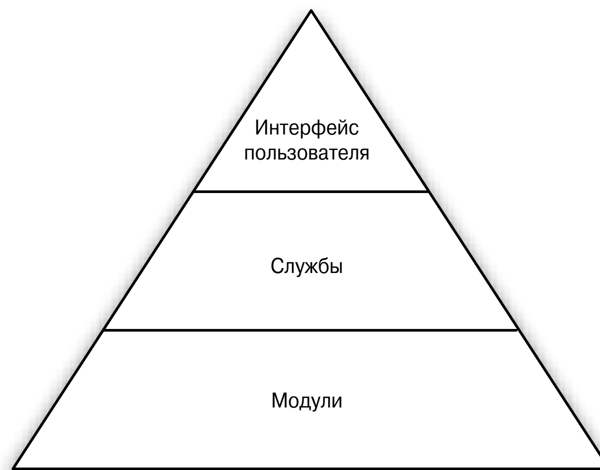

Рис. 7.3. Пирамида тестов

Разработать тесты, выполняющиеся быстро, иногда бывает непросто. Несколько методик их создания приведены в следующих разделах. Большинство представленных методик преследуют единственную цель: минимизировать диапазон покрытия конкретного теста, сфокусировав его на одном аспекте системы. Для этого, в частности, выполняющиеся тесты не должны затрагивать файловую систему, базы данных, библиотеки, инфраструктуру и внешние системы. Все обращения к средам должны быть заменены тес-

товыми двойниками (см. главу 4). На тему разработки через тестирование и модульного тестирования написано достаточно много, поэтому в следующих разделах мы лишь вкратце коснемся этого вопроса.

Избегайте пользовательского интерфейса

Пользовательский интерфейс — это место, в котором пользователи быстрее всего находят ошибки. В результате разработчики непроизвольно сосредотачивают усилия на его тестировании, часто в ущерб тестам других типов.

Однако мы рекомендуем структурировать тесты фиксации таким образом, чтобы они вообще не затрагивали пользовательский интерфейс. Трудности тестирования интерфейса обусловлены двумя причинами. Во-первых, в работу интерфейса вовлечено много компонентов и уровней приложения. Нужно приложить много усилий и затратить много времени, чтобы собрать и подготовить все части приложения для тестирования. Во-вторых, пользовательские интерфейсы приспособлены для работы в "ручном режиме", поэтому создание для них автоматических тестов — трудоемкая задача.

Если структура проекта или применяемая технология позволяют избежать обеих указанных трудностей, иногда может иметь смысл создавать модульные тесты, работающие посредством пользовательского интерфейса, однако наш опыт свидетельствует о том, что тестирование пользовательского интерфейса почти всегда лучше отложить до более поздней стадии конвейера развертывания — стадии приемочного тестирования.

Подходы к тестированию пользовательских интерфейсов подробно рассматриваются в главе 8, посвященной автоматическим приемочным тестам.

Используйте внедрение зависимостей

Внедрение зависимостей (dependency injection), или *инверсия управления* (inversion of control), — это шаблон разработки, определяющий, как отношения между объектами должны устанавливаться извне, а не изнутри объектов. Естественно, данная методика применима только при использовании объектно-ориентированного языка.

Рассмотрим следующий пример. Во время разработки класса `Car` (Автомобиль) его можно структурировать таким образом, чтобы он автоматически порождал собственный объект `Engine` (Двигатель) при каждом создании своего экземпляра. Альтернативный подход заключается в разработке класса `Car` таким образом, чтобы при каждом создании своего экземпляра он вынуждал предоставлять ему объект `Engine`.

Второй, альтернативный подход — пример внедрения зависимости. Это более гибкий подход, потому что он позволяет создавать объекты `Car` с разными типами объектов `Engine`, не изменяя код класса `Car`. Можно даже создать объект `Car` со специальным объектом `TestEngine`, который имитирует реальный объект `Engine` во время тестирования объекта `Car`.

Данная методика — не только хороший способ улучшения гибкости и модульности приложения, она также существенно облегчает ограничение диапазона тестирования классами, которые нужно протестировать в данный момент, позволяя избежать тестирования зависимостей.

Не обращайтесь к базам данных

Разработчики, не имеющие опыта написания автоматических тестов для систем непрерывной интеграции, часто создают тесты, взаимодействующие со слоями кода и про-

веряющие результаты, сохраняемые кодом в базе данных. Это очень простой и естественный подход, но для тестов фиксации он неэффективен.

В первую очередь, такие тесты медленно выполняются. Кроме того, сам тест тяжело проверить, потому что при его повторении результаты могут отличаться. Сложность инфраструктуры приложения может существенно затруднить установку тестов и управление ими. И наконец, если тяжело исключить базу данных из теста, значит, разбиение кода на слои плохое, поскольку не позволяет разделить функции приложения. Это еще один пример того, как требования пригодности к тестированию оказывают давление на команду, вынуждая ее создавать лучший код.

Модульные тесты, составляющие основную часть тестов фиксации, не должны затрагивать базы данных. Для этого нужно иметь возможность отделять тестируемый код от хранилищ. Отделение кодов от хранилищ достигается в результате правильного разбиения кодов на слои, применения эффективных методик, таких как внедрение зависимостей (см. выше), и, в крайнем случае, создания образов баз данных в памяти.

Тем не менее в набор тестов фиксации рекомендуется включить один или два простых дымовых теста. Это должны быть сквозные тесты из набора приемочного тестирования, проверяющие ценные или часто используемые функции.

Избегайте асинхронности в модульных тестах

Асинхронное поведение в диапазоне одного теста затрудняет тестирование системы. Простейший подход к решению этой проблемы заключается в разбиении тестов таким образом, чтобы один тест выполнялся до точки ветвления потока, а другие тесты — после нее.

Например, если система передает сообщения, а затем реагирует на них, заключите процедуру создания и передачи сообщений в оболочку с собственным интерфейсом. Тогда в одном тесте вы сможете проверить вызов с помощью подставного объекта или простой заглушки, реализующей интерфейс сообщения, а в другой тест можно добавить проверку поведения обработчика сообщения, вызвав функцию, которую в рабочем режиме вызывает инфраструктура сообщения. Однако иногда, в зависимости от архитектуры, реализация данного сценария может оказаться весьма трудоемкой.

Мы рекомендуем приложить значительные усилия, направленные на устранение асинхронности в тестах фиксации. Тесты, полагающиеся на инфраструктуру (например, инфраструктуру сообщений, даже если она резидентная), относятся к компонентным, а не к модульным тестам. Более сложные и медленно выполняющиеся компонентные тесты должны быть частью стадии приемочного тестирования, а не фиксации.

Используйте тестовые двойники

Идеальный модульный тест сконцентрирован на небольшом фрагменте кода, обычно на одном классе или нескольких тесно связанных классах. Структура приложения должна способствовать этому. В хорошо структурированной системе каждый класс небольшой и решает свои задачи, взаимодействуя с другими классами. Главный принцип хорошей инкапсуляции состоит в том, что каждый класс “хранит свои секреты” от других классов.

Проблема заключается в том, что в такой хорошо структурированной модульной системе тестирование объекта, работающего в сети отношений с другими объектами, может потребовать громоздкой настройки всех окружающих классов. Типовое решение — имитация взаимодействия с зависимыми и влияющими классами.

Установка заглушек вместо реальных зависимостей — давняя традиция. Мы уже упоминали о внедрении зависимостей (dependency injection) и привели простой пример заглушки `TestEngine`, подставляемой вместо объекта `Engine`.

Заглушка — это замена части системы ее имитационной версией, которая генерирует предопределенные ответы. Заглушки не реагируют на запросы, не предусмотренные при их создании. Это мощный и гибкий подход, полезный на любом уровне — от замены одного простого класса, от которого зависит тест, до замены целой системы.

Использование заглушек для замены подсистемы сообщений

Однажды Дейв работал над торговой системой, которая должна была посредством очереди сообщений взаимодействовать сложным образом с другой системой, разрабатываемой сторонней командой. Коммуникация между системами была богатой и разнообразной. Использовалась коллекция сообщений, которая в значительной степени определяла жизненный цикл сделки и была синхронизирована с обеими системами. Без внешней системы наша система не владела полным жизненным циклом сделки, поэтому нам тяжело было создать сквозной приемочный тест.

Мы реализовали в меру сложную заглушку, имитировавшую операцию в "живой" системе. Заглушка оказалась очень полезной. В частности, она позволила заполнить разрыв в жизненном цикле нашей системы в режиме тестирования. Кроме того, она позволила имитировать тяжелые, граничные ситуации, воспроизвести которые в реальной системе было бы нелегко. И наконец, она позволила разбить зависимости и организовать параллельную разработку систем.

Благодаря заглушке мы вместо поддержки сложной сети взаимодействующих подсистем получили возможность выбора вариантов, когда система должна взаимодействовать с реальным миром, а когда — с простой заглушкой. Развертывание заглушки управлялось конфигурированием, поэтому выбор вариантов осуществлялся путем несложного изменения конфигурационных параметров.

Рекомендуем использовать заглушки главным образом для имитации больших компонентов и подсистем. Для замены компонентов на уровне языка программирования они менее полезны; в этом случае часто предпочтительнее подставные объекты.

Подставные объекты, или Mock-объекты, — сравнительно новая технология. Их использование мотивируется любовью к заглушкам и желанием широко применять, но без написания длинных кодов заглушек. Как хорошо было бы, если бы вместо длинной и скучной реализации в заглушке всех зависимостей тестируемых классов можно было приказать компьютеру создать нужную заглушку автоматически.

Подставной объект именно так и делает. Для создания подставных объектов на рынке программных продуктов есть ряд инструментов, таких как Mockito, Rhino, EasyMock, JMock, NMock, Mocha и др. Они позволяют как бы сказать компьютеру: "Создай мне объект, делающий вид, будто он порожден классом X".

Более того, эти инструменты позволяют с помощью нескольких простых утверждений задать ожидаемое поведение тестируемого кода. В этом состоит важное различие между подставными объектами и заглушками. Заглушке безразлично, как она вызывается, а подставной объект может проверить способ взаимодействия с тестируемым кодом.

Вернемся к примеру с классом `Car` (Автомобиль) и рассмотрим два подхода к его тестированию. Предположим, при вызове метода `Car.drive` (автомобиль.ехать) сначала должен быть вызван метод `Engine.start` (двигатель.завести), а затем — `Engine.accelerate` (двигатель.нажать_на_газ).

Как описывалось выше, в обоих случаях для ассоциирования объекта Car с Engine применим внедрение зависимостей. Упрощенный код классов может выглядеть следующим образом.

```
class Car {
  private Engine engine;

  public Car(Engine engine) {
    this.Engine = engine;
  }

  public void drive() {
    engine.start();
    engine.accelerate();
  }
}

Interface Engine {
  public start();
  public accelerate();
}
```

При использовании заглушки нужно создать тестовую реализацию TestEngine, которая фиксирует факт вызова методов Engine.start и Engine.accelerate. Поскольку требуется, чтобы метод Engine.start был вызван первым, в заглушке нужно, видимо, сгенерировать исключение или каким-либо образом зафиксировать ошибку, если первым вызывается метод Engine.accelerate.

Тест должен создать объект Car, передать в его конструктор объект TestEngine, вызвать метод Car.drive и подтвердить факт правильной последовательности вызовов Engine.start и Engine.accelerate.

```
class TestEngine implements Engine {
  boolean startWasCalled = false;
  boolean accelerateWasCalled = false;
  boolean sequenceWasCorrect = false;

  public start() {
    startWasCalled = true;
  }
  public accelerate() {
    accelerateWasCalled = true;
    if (startWasCalled == true) {
      sequenceWasCorrect = true;
    }
  }
  public boolean wasDriven() {
    return startWasCalled && accelerateWasCalled && _
           sequenceWasCorrect;
  }
}

class CarTestCase extends TestCase {
  public void testShouldStartThenAccelerate() {
    TestEngine engine = new TestEngine();
    Car car = new Car(engine);

    car.drive();

    assertTrue(engine.wasDriven());
  }
}
```

Теперь рассмотрим эквивалентный тест на основе подставных объектов, созданных с помощью инструмента JMock. Подставной объект `Engine` создается путем вызова подставного класса и передачи ссылки на класс или интерфейс, определяющий интерфейс `Engine`.

Необходимо объявить два требования, задающие ожидаемую последовательность вызовов `Engine.start` и `Engine.accelerate`. И наконец, прикажем системе проверить, действительно ли произошло то, что должно было произойти.

```
import jmock;

class CarTestCase extends MockObjectTestCase {
  public void testShouldStartThenAccelerate() {
    Mock mockEngine = mock(Engine);
    Car car = new Car((Engine)mockEngine.proxy());

    mockEngine.expects(once()).method("start");
    mockEngine.expects(once()).method("accelerate");

    car.drive();
  }
}
```

Пример основан на использовании открытого инструмента JMock. Другие инструменты работают аналогично. В данном случае окончательная верификация задается неявно в конце каждого тестового метода.

Преимущества подставных объектов очевидны. Объем кода заметно меньше даже в таком простом примере. В реальных задачах подстановка позволяет сэкономить намного больше усилий. Кроме того, подстановка — прекрасный способ изоляции кодов сторонних производителей от тестов. Вместо кодов сторонних производителей можно подставлять любые интерфейсы, устраняя их таким образом из тестов, что особенно полезно при взаимодействии с дорогостоящими удаленными системами или "тяжеловесными" инфраструктурами.

По сравнению со сборкой всех зависимостей и ассоциированных с ними состояний, тесты, в которых применяются подставные объекты, выполняются очень быстро. Мы настоятельно рекомендуем использовать методики на основе подставных объектов для создания тестов фиксации.

Минимизируйте состояния в тестах

В идеале модульные тесты должны быть сосредоточены на проверке поведения системы. Наиболее общая проблема, особенно для новичков в области автоматического тестирования, — увеличение сложности состояний. Тест можно создать почти для любой формы, в которую пользователь вводит значения для компонента системы и получает результаты. Тест создается путем организации подходящей структуры данных таким образом, чтобы можно было подать ее на вход формы и сравнить результат с ожидаемым. Фактически, все тесты основаны на этом принципе. Проблема состоит в том, что при усложнении системы тестовая структура данных усложняется значительно быстрее.

Реализуя данный подход, вы будете создавать все более сложные тестовые структуры данных, которые тяжело понять и поддерживать. Идеальный тест должен быстро выполняться и легко устанавливаться (и еще легче отбрасываться). С хорошо факторизованным кодом должны ассоциироваться лаконичные тесты. Причиной громоздкости и сложности тестов может быть несовершенная структура системы.

Идеального решения данной проблемы не существует. Мы рекомендуем прилагать усилия, направленные на минимизацию зависимости тестов от состояния системы. Устранить ее полностью невозможно, однако можно контролировать сложность тестовых сред в разумных пределах. Если тесты стали слишком сложными, значит, скорее всего, нужно пересмотреть структуру кода.

Подделывайте время

В автоматических тестах время может породить проблемы по нескольким причинам. Иногда систему нужно запустить в конце рабочего дня в 20:00. Иногда нужно подождать полсекунды, прежде чем перейти к следующему этапу. В некоторых случаях нужно сейчас выполнить операцию таким образом, будто она выполняется 29 февраля следующего года.

Во всех этих случаях попытка привязать тесты к реальному времени может привести стратегию модульного тестирования к полному краху.

При тестировании поведения, зависящего от времени, мы рекомендуем создать для информации о времени специальный слой абстракции — отдельный класс, находящийся под полным вашим контролем. Обычно полезно применить внедрение зависимости, добавляя оболочку для поведения, зависящего от времени, на уровне системы.

Например, с помощью заглушки или подставного объекта можно использовать поведение класса `Clock` (часы) или любой другой выбранной вами абстракции для имитации времени. Если для теста нужно, чтобы сейчас был следующий год или наше время отличалось от реального на полсекунды, сделать это будет несложно.

Имитировать время особенно важно в быстрых сборках, когда поведение требует задержки или ожидания. Чтобы добиться высокой производительности тестирования, структурируйте код таким образом, чтобы во время выполнения теста длительность всех ожиданий была равна нулю. Если для модульного теста нужна реальная задержка, реструктурируйте код или тест, чтобы избежать ее.

Абстрагирование времени у нас уже вошло в привычку. Когда мы пишем код, обращающийся ко времени, мы, не задумываясь, абстрагируем доступ к службам системного времени, а не обращаемся к ним непосредственно в деловой логике.

Применяйте метод “грубой силы”

Разработчики всегда пытаются уменьшить длительность цикла фиксации. Однако в реальности его нужно балансировать со способностью теста идентифицировать на стадии фиксации как можно большее количество ошибок. Нахождение баланса выполняется методом проб и ошибок. Иногда лучше смириться с медлительностью стадии фиксации, чем тратить время на оптимизацию баланса скорости и количества обнаруживаемых ошибок, однако отказ от оптимизации допустим только при небольшой длительности тестов фиксации.

Мы считаем, что стадия фиксации должна выполняться не более десяти минут. Для нее это верхняя допустимая граница. В идеале она должна выполняться менее пяти минут. Однако в больших проектах и десять минут может оказаться недостижимым идеалом. В некоторых командах десять минут считается допустимым компромиссом. Мы считаем это число (десять минут) полезным ориентиром. Когда эта граница нарушена, разработчики начинают делать две вещи, пагубно влияющие на проект. Во-первых, они начинают реже регистрировать изменения, и, во-вторых, если стадия фиксации длится значительно больше десяти минут, они перестают обращать внимание на результаты тестов фиксации.

Существуют два трюка, позволяющих уменьшить время выполнения тестов фиксации. Первый заключается в разбиении набора тестов на несколько групп и параллельном выполнении групп на нескольких компьютерах. Современные серверы непрерывной интеграции поддерживают гриды сборок, что существенно облегчает применение этого трюка. Не забывайте, что компьютерное время дешевое, а человеческое — дорогое. Своевременная обратная связь — значительно более ценный ресурс, чем несколько компьютеров. Второй трюк — метод грубой силы — заключается в удалении из стадии фиксации тестов, выполняющихся слишком долго, и переносе их на стадию приемочного тестирования. Однако учитывайте, что цикл обратной связи по многим ошибкам существенно удлиняется, и вы дольше будете оставаться в неведении о том, что зарегистрированное изменение разрушило систему.

Резюме

Стадия фиксации должна быть сфокусирована на единственной задаче: как можно более быстром обнаружении ограниченного набора наиболее распространенных ошибок, введенных при изменении кода. Естественно, стадия фиксации должна известить разработчика о результате, чтобы он мог либо быстро исправить ошибку, либо перейти к следующей задаче. Ценность обратной связи, предоставляемой стадией фиксации, настолько велика, что всегда имеет смысл потратить время и деньги на повышение ее эффективности и быстродействия.

В конвейере развертывания стадия фиксации выполняется каждый раз, когда кто-либо вносит изменение в код или конфигурацию приложения. Следовательно, она выполняется много раз ежедневно каждым членом команды разработки. Чтобы это было возможным, сборки и тесты должны выполняться быстро. Как только длительность стадии фиксации превысит пять минут, разработчики начнут жаловаться на то, что с системой тяжело работать. Не останьтесь глухими к их жалобам и сделайте все возможное, чтобы уменьшить ее длительность. Но при этом не забывайте о том, что стадия фиксации должна выявить как можно больше ошибок, потому что их обнаружение на более поздних стадиях сделает их устранение намного более дорогим.

Создание и установка стадии фиксации — автоматического процесса, который запускается при каждом изменении, выполняет сборку двоичных кодов, выполняет автоматические тесты и генерирует метрики, — минимум того, что нужно сделать на пути к реализации системы непрерывной интеграции. Реализация стадии фиксации — огромный шаг вперед в направлении повышения качества приложения и надежности процесса поставки. Она вынуждает разработчиков придерживаться многих полезных концепций непрерывной интеграции, таких как регулярная регистрация изменений и немедленное устранение обнаруженных дефектов. Стадия фиксации — лишь начало конвейера развертывания, тем не менее эффект от ее реализации огромен. Изменяется парадигма разработки: когда изменение разрушает систему, вы немедленно получаете сообщение об этом и можете быстро вернуть ее в работоспособное состояние.