5 Разработка через тестирование

Методология разработки через тестирование, или TDD (Test Driven Development), появилась в нашей отрасли уже более 10 лет. Изначально она применялась на волне экстремального программирования (XP, eXtreme Programming), но с тех пор была принята на вооружение Scrum и практически всеми остальными гибкими (Agile) методологиями. Даже группы, не использующие гибкие методологии, применяют TDD.

Когда в 1998 году я впервые услышал о «упреждающем тестировании», я отнесся к нему скептически. Да и кто бы поступил иначе? Кто *начинает* работу с написания модульных тестов? Кто будет делать подобные глупости?

Но к тому времени у меня был уже 30-летний опыт профессионального программирования; я видел, как в отрасли появляются и исчезают новые идеи. Я прекрасно понимал, что ничего не стоит отвергать заранее, особенно если рекомендует такой человек, как Кент Бек.

Так в 1999 году я отправился в Медфорд, штат Орегон, чтобы встретиться с Кентом и научиться у него новой методологии. Результат был просто поразительным!

Мы с Кентом сели у него в офисе и начали программировать простую задачу на Java. Я хотел просто написать свой примитивный код, но Кент воспротивился и провел меня по всему процессу шаг за шагом. Сначала он написал крошечную часть модульного теста, которую и кодом-то нельзя было назвать. Затем он написал код, достаточный для того, чтобы тест компилировался. Затем он написал еще один тест и еще немного кода.

Такой рабочий цикл полностью противоречил всему моему опыту. Я привык писать код не менее часа, прежде чем пытаться откомпилировать или запустить его. Но Кент буквально выполнял свой код каждые 30 секунд или около того. Это было невероятно! Но самое интересное, что этот рабочий цикл был мне знаком! Я сталкивался с ним много лет назад, когда еще ребенком[1] программировал игры на интерпретируемых языках вроде Basic или Logo. В этих языках не было сборки как таковой: вы просто добавляли строку кода и запускали программу. Рабочий цикл проходил очень быстро. И по этой причине программирование на этих языках бывало *очень* производительным.

Но в *настоящем* программировании такой рабочий цикл казался абсурдным. В настоящем программировании вы тратили много времени на написание кода, а потом еще больше времени на то, чтобы заставить его компилироваться. И еще больше времени на отладку. Я ведь *был программистом C++, черт побери*! А в C++ процессы сборки и компоновки могли длиться минутами, а то и часами. Тридцатисекундные рабочие циклы казались немыслимыми. Тем не менее передо мной сидел Кент, который писал свою программу на Java с 30-секундными циклами — и без малейшего намека на то, что работа замедлится. И тогда до меня дошло, что эта простая методология позволяет программировать на настоящих языках с продолжительностью рабочего цикла, типичной для Logo! И я капитально «подсел» на TDD!

---

[1] С высоты своего возраста я считаю ребенком всех, кому меньше 35 лет. Когда мне было за 20, я тратил довольно много времени на написание глупых игр на интерпретируемых языках. Я программировал космические «стрелялки», приключенческие игры, имитаторы скачек, «змейки», азартные игры... и т. п.

Вердикт вынесен

С того времени я узнал, что TDD — нечто много большее, чем простой трюк для сокращения рабочего цикла. Методология обладает множеством преимуществ, которые будут описаны ниже.

Но сначала я должен сказать следующее.

- Вердикт вынесен!
- Дебаты завершены.
- Команда GOTO вредна.
- И TDD работает.

Да, за прошедшие годы о TDD было написано много противоречивых статей и блогов. На первых порах встречалась серьезная критика и сомнения, но в наши дни все дискуссии завершены. Кто бы что ни говорил, TDD работает.

Я знаю, что это утверждение кажется слишком жестким и односторонним, но в конце концов, хирургам уже не нужно доказывать полезность мытья рук. И я не думаю, что программистам нужно защищать TDD.

Как можно называть себя профессионалом, если вы не *знаете*, что весь ваш код работает? А как можно знать, что весь ваш код работает, если вы не тестируете его при каждом внесении изменений? А как тестировать код при каждом внесении изменений, не имея автоматизированных модульных тестов с очень высоким покрытием? Но можно ли создать автоматизированные модульные тесты с очень высоким покрытием без применения TDD?

Впрочем, последнее предложение стоит рассмотреть более подробно. Что же это, собственно, такое — TDD?

Три закона TDD

1. Новый рабочий код пишется только после того, как будет написан модульный тест, который не проходит.
2. Вы пишете ровно такой объем кода модульного теста, какой необходим для того, чтобы этот тест не проходил (если код теста не компилируется, считается, что он не проходит).

3. Вы пишете ровно такой объем рабочего кода, какой необходим для прохождения модульного теста, который в данный момент не проходит.

Эти три закона заставляют вас использовать рабочий цикл продолжительностью около 30 секунд. Сначала вы пишете маленькую часть модульного теста. За эти считанные секунды вы упоминаете в коде имя класса или функции, которые еще не были написаны; естественно, модульный тест не компилируется. Следовательно, далее вы должны написать рабочий код, с которым тест откомпилируется. Но писать больше кода нельзя, поэтому вы переходите к написанию дополнительного кода модульного теста.

Цикл прокручивается снова и снова. Добавляем небольшой фрагмент в тестовый код. Добавляем небольшой фрагмент в рабочий код. Два кодовых потока растут одновременно, превращаясь во взаимодополняющие компоненты. Соответствие между тестами и рабочим кодом напоминает соответствие между антителом и антигеном.

Длинный перечень преимуществ

Уверенность

Разработчик, принявший TDD как профессиональную методологию, пишет десятки тестов каждый день, сотни тестов каждую неделю, тысячи тестов каждый год. И все эти тесты постоянно находятся «под рукой» и запускаются при каждом внесении в код каких-либо изменений.

Я являюсь основным автором и ответственным за сопровождение FitNesse[1] — системы приемочного тестирования на базе Java. На момент написания книги код FitNesse состоял из 64 000 строк, из которых 28 000 содержались в 2200 отдельных модульных тестах. Эти тесты обеспечивают покрытие по меньшей мере 90% рабочего кода[2], а их выполнение занимает около 90 секунд.

Каждый раз, когда я изменяю какую-либо часть FitNesse, я запускаю модульные тесты. Если они проходят, то я практически полностью уверен, что изменения ничего не нарушили. Насколько «практически полностью»? Достаточно, чтобы опубликовать обновленную версию!

---

[1] http://fitnesse.org

[2] 90% — минимальная оценка. На самом деле значение намного выше. Точную величину покрытия трудно рассчитать, потому что программные инструменты «не видят» код, выполняемый во внешних процессах или блоках catch.

Весь процесс контроля качества FitNesse сводится к команде `ant release`. Эта команда собирает FitNesse «с нуля», а затем запускает все модульные и приемочные тесты. Если все тесты проходят успешно, я публикую результат.

Снижение плотности дефектов

FitNesse не является критически важным приложением. Если в FitNesse закрадется ошибка, никто не умрет и никто не потеряет миллионы долларов. Исходя из этого, я могу себе позволить опубликовать новую версию на основании только прохождения тестов. С другой стороны, у FitNesse тысячи пользователей, и при том, что за последний код кодовая база расширилась на 20 000 строк, мой список дефектов состоит только из 17 позиций (многие из которых имеют чисто косметическую природу). Таким образом, я знаю, что плотность дефектов в FitNesse чрезвычайно низка.

И этот эффект не уникален. Существенное снижение количества дефектов при использовании TDD описано в ряде отчетов[1] и исследований[2]. От IBM до Microsoft, от Sabre до Symantec — компании и группы сообщают о снижении количества дефектов в 2, 5 и даже 10 раз. Настоящий профессионал не может игнорировать такие показатели.

Смелость

Почему мы не исправляем плохой код сразу же, как только увидим его? Наша первая реакция на неаккуратно написанную, запутанную функцию: «Ну и мешанина, надо бы исправить». Вторая реакция: «Пусть это сделает кто-нибудь другой!» Почему? Потому что вы знаете:

---

[1] http://www.objectmentor.com/omSolutions/agile_customers.html

[2] E. Michael Maximilien, Laurie Williams, “Assessing Test-Driven Development at IBM,“ http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF

B. George, and L. Williams, “An Initial Investigation of Test-Driven Development in Industry,“ http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,“ *IEEE Computer*, Volume 38, Issue 9, pp. 43–50.

Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, “Realizing quality improvement through test driven development: results and experiences of four industrial teams,“ Springer Science + Business Media, LLC 2008: http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

притронувшись к коду, вы рискуете его «сломать»; а если код будет сломан, то он автоматически переходит под вашу ответственность.

А если вы твердо уверены, что чистка кода ничего не нарушит? Если вы просто нажимаете кнопку и через 90 секунд узнаете, что изменения ничего не нарушили, а принесли только пользу?

Это одно из величайших преимуществ TDD. Если у вас имеется пакет тестов, которому можно доверять, вы перестаете бояться вносить изменения. Видя плохой код, вы просто чистите его «на месте». Код становится глиной, из которой лепятся простые, эстетичные структуры.

Когда программист перестает бояться чистить код, он чистит его! Чистый код проще понять, проще изменять и проще расширять. С упрощением кода вероятность дефектов становится еще ниже. Происходит стабильное улучшение кодовой базы — вместо «загнивания кода», столь привычного для нашей отрасли. Разве профессиональный программист может допустить, чтобы загнивание продолжалось?

Документация

Вы когда-нибудь использовали сторонние библиотеки? Фирма-разработчик обычно присылает красивое руководство с десятками глянцевых иллюстраций с кружочками и стрелочками; на обратной стороне каждой иллюстрации приводится абзац текста с описанием настройки, развертывания и других операций с этой библиотекой. А в самом конце, где-нибудь в приложении, прячется маленький невзрачный раздел с примерами кода.

Куда вы первым делом заглянете в таком руководстве? Если вы программист, то вы обратитесь к примерам кода. Вы сделаете это, потому что знаете: код расскажет всю правду. Цветные глянцевые иллюстрации с кружочками и стрелочками выглядят очень мило, но если вы хотите узнать, как использовать код, необходимо читать код.

Каждый модульный тест, написанный с соблюдением трех законов, представляет собой пример использования системы, сформулированный в виде программного кода. Если вы выполняли три закона, то в вашем коде будет модульный тест, описывающий создание каждого объекта в системе, для каждого способа создания таких объектов. В нем будет модульный тест, описывающий вызов каждой функции в системе, для каждого осмысленного способа вызова. Для каждой операции, по поводу которой у вас могут возникнуть вопросы, будет модульный тест, подробно описывающий ее выполнение.

Модульные тесты представляют собой документы, описывающие самый нижний архитектурный уровень системы. Они однозначны, точны, написаны на языке, понятном для аудитории, и достаточно точны и формальны для выполнения. Это самая лучшая низкоуровневая документация, которая только возможна. Какой профессионал не захочет создать такую документацию?

Архитектура

Если вы соблюдаете три закона и пишете тесты раньше рабочего кода, вы сталкиваетесь с дилеммой. Часто вы точно знаете, какой код нужно написать, но три закона приказывают сначала написать модульный тест, который не пройдет, потому что код еще не существует! Таким образом, вам приходится думать о тестировании еще не написанного кода.

Проблема с тестированием кода заключается в необходимости изоляции этого кода. Часто бывает трудно тестировать функцию, вызывающую другие функции. Чтобы написать такой тест, необходимо каким-то образом отделить функцию от всех остальных. Иначе говоря, необходимость тестирования заставляет вас продумать хорошую архитектуру приложения.

Если вы не начинаете с написания тестов, то ничто не помешает вам свалить все функции в одну кучу, не поддающуюся тестированию. Если тесты пишутся позднее, возможно, вам удастся протестировать входное и выходное поведение этой кучи, но, скорее всего, с тестированием отдельных функций возникнут большие проблемы.

Таким образом, соблюдение трех законов и опережающее написание тестов приводит к более качественной архитектуре с меньшим количеством привязок. Какой профессионал откажется от инструментов, применение которых приводит к совершенствованию архитектуры?

«Но я могу написать тесты позднее», скажете вы. Нет, не можете. Конечно, *некоторые* тесты можно написать позднее. Можно даже обеспечить высокое покрытие, если вы проследите за его измерением. Однако тесты, написанные позднее, лишь *защищают* от ошибок — тогда как тесты, написанные с опережением, их активно *атакуют*. Тесты, написанные позднее, пишутся разработчиком, который уже сформировал код и знает, как решалась задача. Такие тесты никак не сравнятся по полноте и актуальности с тестами, написанными заранее.

Выбор профессионалов

Из всего сказанного следует, что TDD — выбор профессионалов. Эта методология повышает уверенность, придает смелости разработчикам, снижает количество дефектов, формирует документацию и улучшает архитектуру. При таком количестве доводов в пользу TDD отказ от использования этой методологии можно считать проявлением непрофессионализма.

Чем TDD не является

При всех своих достоинствах TDD — не религия и не панацея. Выполнение трех законов не гарантирует ни одного из перечисленных преимуществ. Плохой код можно написать даже при предварительном написании тестов. Да и сами тесты тоже могут быть написаны плохо.

В некоторых ситуациях три закона оказываются просто непрактичными или неподходящими. Такие ситуации встречаются редко, но они все же возможны. Ни один профессиональный разработчик не станет применять методологию, которая в конкретной ситуации приносит больше вреда, чем пользы.