ОГЛАВЛЕНИЕ

Смелость

Пока мы видели, что если следовать трем правилам разработки через тестирование, то у нас появляется много преимуществ: меньше отладки, качественная низкоуровневая документация, радость и разделение связей. Но это лишь сопутствующие преимущества, ни одно из них не является главной причиной применения разработки через тестирование. Настоящая причина — это воспитание смелости.

Я уже рассказывал историю в самом начале книги, но стоит ее повторить.

Представьте, что вы смотрите на некий уже написанный код на экране. Там бардак. Первая мысль, которая приходит в голову: «Нужно почистить его». Но следующая мысль будет примерно такой: «Нет, я в это не полезу!» Вы думаете, что если влезть в код, то он перестанет работать. А если он перестанет работать, вина ваша. Поэтому вы отстраняетесь от кода подальше, оставляя его гнить и чахнуть.

В вас говорит страх. Вы боитесь кода. Боитесь что-либо делать с ним. Боитесь что-то сломать, потому что будут последствия. Так вы отказываетесь от того единственного, что может улучшить код, — от его чистки.

Если в команде каждый придерживается такого поведения, код будет портиться. Никто не возьмется его почистить. Никто не улучшит его. Каждая новая функция будет добавлена таким образом, чтобы свести на нет непосредственный риск для программистов. Будут добавлены связи и дубликаты, потому что они уменьшают непосредственный риск, даже если нарушают структуру и целостность кода.

В конце концов код становится чудовищно запутанным, как спагетти, его невозможно сопровождать, работа над таким кодом едва ли будет продвигаться. Сложность задач будет расти в геометрической прогрессии. Менеджеры в отчаянии. Они будут нанимать все больше программистов в надежде улучшить производительность, но улучшение не будет достигнуто.

Наконец, достигнув критической точки, руководство согласится на требование программистов переписать всю программу с самого начала. И начинается то же самое.

Представьте себе другой сценарий. Вернемся к монитору, на котором мы видим запутанный код. Первая мысль, которая вас посещает, — надо почистить код. Что, если бы у вас был настолько полный тестовый набор, что ему можно полностью доверять? А если этот тестовый набор работал бы быстро? Что бы вы подумали следующим делом? Наверное, что-то вроде этого:

> *Боже, думаю, просто надо поменять имя этой переменной. О, код все еще проходит тесты. Ладно, а теперь я разделю ту большую функцию на две поменьше... Здорово, все еще удается пройти... Хорошо, теперь, думаю, можно перенести одну из этих новых функций в другой класс. Опа! Тест не пройден. Так, ну-ка, вернем все... А, я понял, надо было переместить и саму переменную. Да, тест снова пройден...*

Когда у вас есть полный набор тестов, вы больше не боитесь вносить изменения в код. Вы больше не боитесь его чистить. Вы просто возьмете и почистите код. Код будет опрятным и чистым. Структура программы останется неизменной. Вы не будете плодить массу гниющего спагетти, которая вгонит команду в уныние, приводящее к низкой производительности и, в конце концов, к провалу.

Поэтому мы применяем разработку через тестирование. Мы применяем этот метод, потому что он вселяет в нас смелость поддерживать код в чистоте и порядке. Смелость вести себя профессионально.

Рефакторинг

Рефакторинг — еще одна тема, достойная целой книги. К счастью, ее уже написал Мартин Фаулер[1]. В этой главе мы просто обсудим это тему, не углубляясь в отдельные методы. И как и прежде, в этой главе нет кода.

Рефакторинг — это метод улучшения структуры кода без изменения его поведения, определенного тестами. Другими словами, мы вносим изменения в имена, классы, функции и выражения, не проваливая никаких тестов. Мы улучшаем структуру программы без воздействия на ее выполнение.

Конечно же, эта дисциплина тесно связана с разработкой через тестирование. Чтобы без опасений перепроектировать код, нужен тестовый набор, который с высокой степенью вероятности укажет нам на то, что мы ничего не испортим.

Изменения, выполненные во время рефакторинга, разнятся от простых косметических до глубокой правки структуры. Такие изменения могут представлять собой просто изменения в названиях или сложную замену операторов switch на полиморфные отправки. Большие функции будут разбиты на те, что поменьше, с более удачными названиями. Списки аргументов будут изменены на

---

[1] *Fowler M.* Refactoring: Improving the Design of Existing Code. 2nd ed. Boston, Massachusetts: Addison-Wesley, 2019.

объекты. Классы с большим количеством методов будут разделены на множество мелких классов. Функции будут перемещены из одного класса в другой. Из классов будут выделены подклассы или внутренние классы. Зависимости будут инвертированы, а модули перемещены через границы архитектуры.

И пока все это происходит, наша программа непременно проходит тесты.

Красный/зеленый/рефакторинг

Ход рефакторинга естественным образом связан с тремя правилами разработки через тестирование приемом «красный/зеленый/рефакторинг» (рис. 5.1).

Рис. 5.1. Цикл «красный/зеленый/рефакторинг»

1. Сначала мы создаем тест, который не получается пройти.
2. Потом пишем код и проходим тест.
3. Затем подчищаем код.
4. Далее возвращаемся к шагу 1.

Написание рабочего кода и написание чистого кода — это две разные вещи. Делать одновременно то и другое необычайно сложно, так как это совершенно разная деятельность.

Довольно тяжело написать рабочий код, не говоря уже о соблюдении его чистоты. Поэтому мы сначала ориентируемся на написание рабочего кода, что бы там ни происходило в головах нашего сумрачного гения. Затем, когда все заработало, мы устраняем беспорядок, который натворили.

Это дает понять, что рефакторинг кода — процесс непрерывный, и его не проводят по плану. Мы не плодим беспорядок несколько дней кряду, чтобы потом долго его подчищать. Мы лучше создадим легкий беспорядок и через минуту-две все исправим.

Слово «рефакторинг» никогда не должно появляться в графике работ. Это не тот род деятельности, который можно провести по плану. Мы не выделяем времени на рефакторинг кода. Рефакторинг — это часть нашей ежеминутной, ежечасной рутины при написании ПО.

Большой рефакторинг

Иногда требования меняются так, что вы осознаете: дизайн и архитектура программы не совсем подходят. Тогда вы решаете внести значительные изменения в структуру программы. Такие изменения вносят в цикле «красный/зеленый/рефакторинг»ь Мы не создаем программы специально для того, чтобы вносить изменения в структуру. Мы не выделяем времени в графике работ на такой глубокий рефакторинг кода. Маленькими порциями мы переносим код, продолжая добавлять новые функции за время обычного цикла Agile.

Такое изменение в структуру программы можно вносить несколько дней, недель или даже месяцев. Все это время программа проходит все необходимые тесты и готова к развертыванию, даже если изменение структуры не полностью завершено.

ПРОСТОТА ПРОЕКТИРОВАНИЯ

Метод «простота проектирования» — одна из целей рефакторинга. Простота проектирования — метод, предполагающий написание только необходимого кода, чтобы сохранять простоту структуры, его небольшой размер и наибольшую выразительность.

Правила простого проектирования Кента Бека.

1. Пройти все тесты.
2. Проявить намерение.
3. Удалить дубликаты.
4. Сократить количество элементов.

Номера пунктов означают порядок действий, в котором эти правила выполняются, и их приоритет.

Пункт 1 говорит сам за себя. Код должен пройти все тесты. Он должен работать.

В пункте 2 указано, что после того как код заработал, ему нужно придать выразительность. Он должен явно отражать намерения программиста. Код нужно писать так, чтобы он легко читался и содержал достаточно сведений. Как раз сейчас мы проводим косметический рефакторинг кода, в течение которого вносим много простых изменений. Нужно также разделить большие функции на мелкие, дав им более простые и понятные названия.

В пункте 3 говорится, что после того как код получился в высшей мере описательным и выразительным, мы старательно выискиваем и удаляем все дубликаты. Не нужно, чтобы в коде повторялось одно и то же. Во время такой деятельности проводить рефакторинг, как правило, сложнее. Иногда удалить дубликаты так же просто, как перенести дублирующийся код в функцию и вызвать его из разных мест. В других случаях требуются решения интереснее, например паттерны проектирования[1]: *метод шаблонов*, *стратегия*, *декоратор* или *посетитель*.

В пункте 4 говорится о том, что как только мы удалили все дубликаты, нужно стремиться уменьшить количество структурных элементов, например классов, функций, переменных и так далее.

Цель метода «простота проектирования» — поддерживать наиболее возможную легковесность кода.

Легковесность

При проектировании программа может получиться как достаточно простой, так и необычайно сложной. Чем сложнее структура, тем больше умственная нагрузка на программиста. Эта умственная нагрузка — вес структуры программы. Чем больше вес программы, тем больше времени и усилий будет затрачено программистами на изучение и управление этой программой.

Таким же образом сложность требований также варьируется от небольших до огромных. Чем сложнее требования, тем больше времени и сил понадобится, чтобы изучить эту программу и управлять ею.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.: ил.