РОЛЬ UNIT-ТЕСТИРОВАНИЯ В ПРОЦЕССЕ ОБУЧЕНИЯ ПРОГРАММИРОВАНИЮ

Л. Ф. Дробушевич, В. В. Конах

Белорусский государственный университет
Минск, Беларусь
E-mail: droblf@bsu.by; konakh@bsu.by

Рассматриваются вопросы обучения тестированию в дисциплинах, связанных с программированием, на примере юнит-тестирования.

The problems of training the testing principles in disciplines related to the programming on the example of unit testing.

Ключевые слова: тест, модуль, юнит-тестирование, экстремальное программирование (XP), методология TDD, P-схема, фреймворк, рефакторинг.

Keywords: test, module, unit testing, extreme programming (XP), the methodology of TDD, R-scheme, the framework refactoring.

Юнит-тестирование, или модульное тестирование, − процесс, позволяющий выявить дефекты в отдельных небольших фрагментах (модулях) программы.

Этот термин пришел из традиционной терминологии тестирования, когда не было объектно-ориентированных программ. Тогда под модулем понималась процедура или функция, а тесты разделялись *по методам черного и белого ящиков* [1]. Например, тест, который запускает программу и проверяет ее код возврата, использует метод черного ящика (функциональное тестирование). Юнит-тесты обычно используют метод белого ящика, так как тесты создаются с учетом доступа к внутренней структуре тестируемого кода.

В объектно-ориентированных программах тестируемый модуль может иметь различные формы в зависимости от контекста: отдельный реализованный метод класса, класс целиком (объект) и даже совокупность классов (в UML модулем может называться компонент, выполняющий какую-то функциональность) [2].

Техники тестирования реализованных методов класса почти ничем не отличаются от классических техник тестирования процедур и функций (могут добавляться критерии тестирования пред- и постусловий методов). Модуль, выбранный для тестирования в объектно-ориентированной программе, чаще всего относится к классу, поэтому используется тестирование протокола класса, т. е. проверка сценариев использования объекта класса [3].

Юнит-тестирование обычно выполняется программистами. Поэтому оно является существенным аспектом при обучении любому языку программирования. Здесь важны два момента.

Первый момент заключается в том, чтобы определить место для обучения тестированию в соответствующих курсах программирования. Второй момент связан с практическим использованием автоматизированного тестирования и обучением современным методологиям разработки приложений. Рассмотрим эти два момента более подробно.

Традиционно программистов учат сначала писать код программы, а затем ее тестировать. Поэтому программисты считают, что прерывание процесса кодирования для создания теста отнимет много времени и они забудут, на чем остановились. Это ошибочная концепция, ибо при написании юнит-тестов становится видно, что очень часто эти тесты определяют приложение. Основная работа выливается в придумывание и написание тестов, а кодирование самого приложения превращается в последовательное добавление кусочков кода для того, чтобы новые тесты проходили. Противники тестирования утверждают, что так писать программы

слишком сложно. Однако если следовать всегда по одному и тому же пути – принять решение по функциональности, написать тест, написать код, чтобы тест проходил, то получается путь, ведущий к лучшему коду.

Таким образом, если приучить студентов к юнит-тестированию на начальном этапе обучения программированию, то они придут к пониманию того, что тест – это не только инструмент для проверки кода на корректность, но и среда, в которой проходит только правильный код. К тому же при разработке приложения можно будет обнаружить, что отдельные модули трудно тестировать из-за их многофункциональности или сложной иерархии классов, соответственно, появится необходимость разбиения или упрощения, что в итоге также улучшит код приложения [4].

Для регулярного и углубленного юнит-тестирования необходима автоматизированная поддержка процесса создания и запуска тестов. В большинстве современных систем программирования имеются средства автоматизации процесса тестирования. Чаще всего юнит-тесты создаются и запускаются с использованием соответствующего фреймворка. Фреймворк дает возможность протестировать исходные коды любых приложений как отдельные изолированные функциональные модули (класс или метод).

Сегодня гибкие (agile) методологии являются наиболее популярными методологиями разработки программного обеспечения (ПО). Особое место здесь занимает экстремальное программирование (XP) [5]. Юнит-тестирование является ключевым моментом в методологии XP. Сегодня эта практика получила название «Разработка, Управляемая Тестами» (Test Driven Development, TDD), также известная как «test-first programming». Следующие два принципа являются базовыми в TDD:

- написание тестов до начала реализации программы (класса, метода);
- полная автоматизация и поддержка процесса тестирования.

XP невозможно практиковать без юнит-тестирования. Юнит-тесты дают уверенность в том, что код работает верно. Тесты также предоставляют дополнительные гарантии в дальнейшем без опасений изменять любой код в программе (в том числе чужой), что важно при командной разработке для всех гибких методологий. Когда разработчики вносят изменения, прогон юнит-тестов дает уверенность в том, что никто не «испортил» код.

TDD – необязательный компонент модульного тестирования. Скорее это дополнительная практика, которую использует модульное тестирование с тех пор, как появились фреймворки для юнит-тестирования практически во всех современных системах программирования.

Если команда разработчиков использует XP с самого начала, каждый класс и метод в классе будет иметь набор автоматических юнит-тестов. Перед тем как внести изменения в код, разработчик запускает все прежние юнит-тесты, чтобы убедиться в их успешном прохождении. Затем он пишет юнит-тесты для нового функционала (или изменений), после чего реализует функционал и запускает все тесты снова. И так до тех пор, пока все тесты не проходят на 100 % успешно.

К сожалению, при обучении программированию, да и в литературе по методам программирования и тестирования, мало уделяется внимания тому, что плохо спроектированный код почти невозможно автоматически тестировать. Для некачественного кода создать юнит-тесты крайне сложно, так как такой код изначально для этого не предназначен. И наоборот, написание тестов до начала реализации вынуждает разработчика проектировать архитектуру программы более аккуратно и максимально просто.

Внедрению модульного тестирования в процесс обучения должно предшествовать обучение планированию и созданию тестов. Сразу встает вопрос: сколько нужно тестов? Тестов должно быть столько, чтобы в будущем программном коде не осталось, например, «непокрытых» ветвей или путей [5].

Большую помощь при создания автоматических тестов могут оказать визуальные средства для записи алгоритмов. Смысл в том, что хороший тестовый метод должен проверять толь-

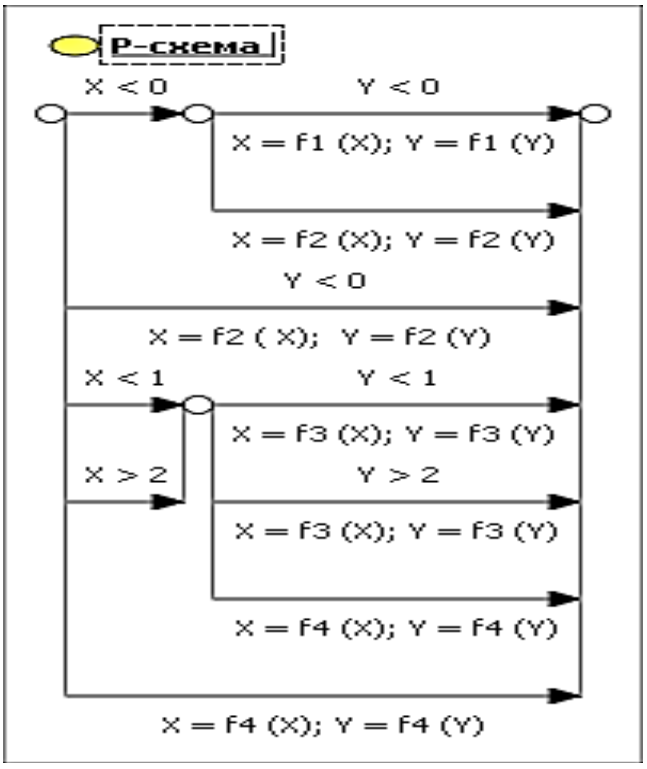
ко один аспект поведения, т. е. он должен содержать только одно тестовое условие. Если условий несколько, необходимо создавать систему тестов, при этом каждое условие проверяется отдельным тестовым методом. Среди визуальных нотаций для записи алгоритмов, по мнению авторов, предпочтение можно отдать графическим Р-схемам [6].

В подтверждение сказанного рассмотрим пример создания юнит-тестов на основе записи алгоритма для логической задачи с функциональными вычислениями в виде графической Р-схемы.

Пример. Создать систему тестов (входные и ожидаемые результаты) для покрытия всех путей в будущей программе. Программа должна читать числа X и Y, вычислять значения по алгоритму и выводить, например, на консоль исходные числа и полученные значения.

Спецификация программы. На входе действительные числа X и Y. Программа вычисляет значения по следующему алгоритму:

- Если X и Y отрицательны, то X заменить на f1 (X), а Y заменить на f1 (Y).
- Если отрицательно только одно из чисел, то X заменить на f2 (X), а Y заменить на f2 (Y).
- Если оба числа неотрицательны и ни одно из них не принадлежит отрезку [1,2], то X заменить на f3 (X), а Y заменить на f3 (Y).
- В остальных случаях X заменить на f4 (X), а Y заменить на f4 (Y).



Р-схема алгоритма

Алгоритм, записанный в виде Р-схемы, показывает, что число необходимых позитивных тестов для «покрытия» всех путей в программе, реализующей данный алгоритм, будет равно девяти. Чтобы создать тесты, достаточно просто «подсобрать» условия на дугах по всем путям Р-схемы алгоритма. Пути выбираем, осуществляя проход в Р-схеме сверху вниз по дугам. И тогда в качестве тестов можно взять следующие:

1. X < 0; Y < 0; ожидаемый результат X = f1 (X); Y = f1 (Y).

2. $X < 0$; $Y \geq 0$; ожидаемый результат X = f2 (X); Y = f2 (Y).
3. $X \geq 0$; $Y < 0$; ожидаемый результат X = f2 (X); Y = f2 (Y).
4. $X \geq 0$ и $X < 1$; $Y \geq 0$ и $Y < 1$; ожидаемый результат X = f3 (X); Y = f3 (Y).
5. $X > 2$; $Y \geq 0$ и $Y < 1$; ожидаемый результат X = f3 (X); Y = f3 (Y).
6. $X \geq 0$ и $X < 1$; $Y > 2$; ожидаемый результат X = f3 (X); Y = f3 (Y).
7. $X \geq 0$ и $X < 1$; $Y > 2$; ожидаемый результат X = f3 (X); Y = f3 (Y).
8. $X \geq 1$ и $X \leq 2$; $Y \geq 0$; ожидаемый результат X = f4 (X); Y = f4 (Y).
9. $X \geq 0$ и $Y \geq 1$ и $Y \leq 2$; ожидаемый результат X = f4 (X); Y = f4 (Y).

Анализируя алгоритм по Р-схеме и созданные тесты до написания кода, видим, что имеем дублирование ожидаемой функциональности – f2, f3, f4. Рефакторинг, который в методологии TDD проводится обязательно после реализации функциональности и «прохода» всех созданных тестов и нацелен, в первую очередь, на устранения дублирования кода. Хотя избежать подобного дублирования функциональности можно было бы уже на стадии проектирования алгоритма.

Основная цель временных затрат на создание как раз и заключается в улучшении качества исходного кода. Многие ошибки обнаруживаются уже на ранних стадиях проектирования алгоритма и непрерывного процесса тестирования при создании нового кода на основе тестов, рефакторинга и безопасного изменения поведения старого кода. Поэтому потенциальная возможность написания юнит-тестов задает, можно сказать, нижнюю планку качества проектирования приложения.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. *Майерс Г.* Искусство тестирования программ М. : Финансы и статистика, 1982.
2. *Арлоу Д., Нейштадт А.* UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование. 2-е изд. СПб. : Символ-Плюс, 2007.
3. *Орлов С. А.* Технология разработки программного обеспечения СПб. : Питер, 2002.
4. *Дробушевич Л. Ф., Конах В. В.* Использование визуальных нотаций для модульного тестирования программ // Междунар. конгресс по информатике: информационные системы и технологии. Минск, 2013.
5. *Кент Бек.* Экстремальное программирование. СПб. : Питер, 2003.
6. *Вельбицкий И. В.* Визуальная технология программирования нового поколения для широкого применения на базе стандарта ISO/IEC 8631 : междунар. конф. Кипр, 2010.