**Модульное тестирование.  
Зачем, как и кто**

[*Сергей Мартыненко*](http://software-testing.ru/lib/martynenko/)

**Терминология**

**Unit testing (юнит тестирование или модульное тестирование)** — заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Поэлементное тестирование — первейшая возможность реализовать исходный код. Оценивая каждый элемент изолированно и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы.

**Unit (Элемент)** — наименьший компонент, который можно скомпилировать.

**Драйверы** — модули тестов, которые запускают тестируемый элемент.

**Заглушки** — заменяют недостающие компоненты, которые вызываются элементом и выполняют следующие действия:

* возвращаются к элементу, не выполняя никаких других действий;
* отображают трассировочное сообщение и иногда предлагают тестеру продолжить тестирование;
* возвращают постоянное значение или предлагают тестеру самому ввести возвращаемое значение;
* осуществляют упрощенную реализацию недостающей компоненты;
* Имитируют исключительные или аварийные условия.

**White-box testing.** Для конструирования тестов используются внутренняя структура кода и управляющая логика. При этом существует вероятность, что код будет проверяться так, как он был написан, а это не гарантирует корректность логики.

**Black-box testing.** Для конструирования тестов используются требования и спецификации ПО. Недостатки:

* таким способом невозможно найти взаимоуничтожающихся ошибок,
* некоторые ошибки возникают достаточно редко (ошибки работы с памятью) и потому их трудно найти и воспроизвести

**Стратегия модульного тестирования**

Модульное тестирование является одной из ключевых практик методологии экстремального программирования. Сторонники XP приводят следующие доводы в защиту этой практики:

* Написание тестов помогает войти в рабочий ритм
* Придает уверенность в работоспособности кода.
* Дает запас прочности при дальнейшей интеграции или изменениях кода.

Согласен, вхождение в рабочий ритм — благородная задача. Уверенность в работоспособности — тоже хорошо. Но «уверенности в работоспособности» я предпочитаю действительно работоспособный код. Пусть даже при этом я не совсем «уверен».

Ключевой фактор при оценке перспективности любого метода — стоимость проекта. Дополнительная работа по созданию тестов, их кодированию и проверке результатов вносит существенный вклад в общую стоимость проекта. И то, что продукт окажется более качественным не всегда перевешивает то, что он будет существенно дороже.

Известно, что продукт оптимальный по набору бюджет/функциональность/качество получается при применении различных способов обеспечения качества. Бездумное применение тотального модульного тестирования почти гарантированно приведет к получению неоптимального продукта. И никакие «запасы прочности» и «быстрый вход в рабочий ритм» не спасут проект от провала.

На мой взгляд, модульное тестирование оправдано, если оно:

* Снижает время на отладку
* Дает возможность поиска ошибок с меньшими затратами, нежели при других подходах
* Дает возможность дешевого поиска ошибок при изменениях кода в дальнейшем

Суммарный выигрыш от применения модульных тестов должен быть больше, чем затраты на их создание и поддержание в актуальном состоянии.

Если в результате исправления ошибок интеграции меняется исходный код, в нем с большой вероятностью появляются ошибки. Если в результате добавления новой функциональности меняется исходный код, в нем с большой вероятностью появляются ошибки. И искать их лучше с помощью ранее созданных модульных тестов.

Цель модульного тестирования:

***Получение работоспособного кода с наименьшими затратами. И его применение оправдано тогда и только тогда, когда оно дает больший эффект, нежели другие методы.***

Отсюда следует несколько выводов:

* Нет смысла писать тесты на весь код. Некоторые ошибки проще найти на более поздних стадиях. Так, например, для ООП данное правило может звучать так: нет смысла писать тесты на класс, который используется только одним классом. Эффективней написать тесты на вызывающий класс и создать тесты тестирующие все участки кода.
* Писать тесты для кода потенциально подверженного изменениям более выгодно, чем для кода, изменение которого не предполагается. Сложная логика меняется чаще, чем простая. Следовательно, в первую очередь имеет смысл писать модульные тесты на сложную логику. А на простую логику писать позднее или вообще тестировать другими методами.
* Для того чтобы как можно реже изменять тесты следует хорошо планировать интерфейсы. То же самое можно сказать и применительно к написанию исходного кода. Действительно, создание хорошей архитектуры часто определяет дальнейший ход проекта. И есть оптимум, на каком этапе архитектура «достаточно хороша». Все так, но я хочу сказать о другом:

***Если в проекте применяется модульное тестирование, то тщательное планирование интерфейсов становится более выгодным. Внедрению модульного тестирования должно предшествовать внедрение планирования интерфейсов.***

**Планирование тестов**

Первый вопрос, который встает перед нами: «Сколько нужно тестов». Ответ, который часто дается: тестов должно быть столько, чтобы не осталось неоттестированных участков. Можно даже ввести формальное правило:

***Код с не оттестированными участками не может быть опубликован***

Проблема в том, что хотя неоттестированный код почти наверняка неработоспособен, но полное покрытие не гарантирует работоспособности. Написание тестов исходя только из уже существующего кода только для того, чтобы иметь стопроцентное покрытие кода тестами — порочная практика. Такой подход со всей неизбежностью приведет к существованию оттестированного, но неработоспособного кода. Кроме того, метод белового ящика, как правило, приводит к созданию позитивных тестов. А ошибки, как правило, находятся негативными тестами. В тестировании вопрос «Как я могу сломать?» гораздо эффективней вопроса «Как я могу подтвердить правильность?». Это наглядно демонстрирует статья [61 тест, который потряс программу](http://forums.software-testing.ru/index.php?automodule=blog&blogid=11&cmd=showentry&eid=195).

В первую очередь тесты должны соответствовать не коду, а требованиям. Правило, которое следует применять:

***Тесты должны базироваться на спецификации.***

Пример такого подхода можно посмотреть в статье [Тривиальная задача](http://forums.software-testing.ru/index.php?automodule=blog&blogid=11&view=showday&d=23&m=1&y=2006).

Один из эффективных инструментов, для определения полноты тестового набора — матрица покрытия.

***На каждое требование должен быть, как минимум, один тест. Неважно, ручной или автоматический.***

При подготовке тестового набора рекомендую начать с простого позитивного теста. Затраты на его создание минимальны. Да вероятность создания кода, не работающего в штатном режиме, гораздо меньше, чем отсутствие обработки исключительных ситуаций. Но исключительные условия в работе программы редки. Как правило, все работает в штатном режиме. Тесты на обработку некорректных условий, находят ошибки гораздо чаще, но если выяснится, что программа не обрабатывает штатные ситуации, то она просто никому не нужна.

***Простой позитивный тест нужен т.к. несмотря на малую вероятность нахождения ошибки, цена пропущенной ошибки чрезмерно высока.***

Последующие тесты должны создаваться при помощи формальных методик тестирования. Таких как, классы эквивалентности, исследование граничных условий, метод ортогональных матриц и т.д.. Тестирование накопило довольно много приемов подготовки тестов и если эти приемы создавались, то видимо было зачем.

Последнюю проверку полноты тестового набора следует проводить с помощью формальной метрики «Code Coverage». Она показывает неполноту тестового набора. И дальнейшие тесты можно писать на основании анализа неоттестированных участков.

***Наиболее эффективный способ создания тестового набора — совместное использование методов черного и белого ящиков.***

**Распределение обязанностей**

Где-то я читал следующую фразу: «Попросите программиста составить для вас (тестера) план тестов». А потом тестер будет кодировать тесты. Генеральный директор рисовать дизайн, а администратор баз данных писать руководство пользователя. Не очень воодушевляющая картина.

Кодировать модульные тесты проще всего программисту, который пишет исходный код. Но перед этим их нужно придумать. Кодирование тестов и разработка тестовых сценариев это две разные задачи. И для последней нужны навыки дизайнера сценариев. Если программист ими обладает, то все прекрасно. Если нет, то имеет смысл поручить это тестировщику. Нужно просто четко определить решаемые задачи и навыки, необходимые для их решения.

|  |  |  |
| --- | --- | --- |
| **Задача** | **Требуемые навыки** | **Роль** |
| Определение методов обеспечения качества ПО | Отличное знание теории тестрования | Ведущий тестировщик проекта |
| Создание тестов | Хорошее знание методов тестирования | Дизайнер тестовых сценариев |
| Кодирование тестов | Средние навыки программирования | Программист автоматических тестов |
| Выполнение тестов | Знание среды выполнения тестов | Тестер |

Вполне возможно, что роль ведущего тестировщика проекта будет выполнять аналитик или менеджер проекта, роль дизайнер тестовых сценариев — программист. А может быть и так, что все эти роли будет выполнять тестировщик.

***Не важно кто конкретно будет выполнять работу, и как будет называться должность. Главное, чтобы сотрудник обладал необходимыми навыками.***