Категории программных тестов

Вы порой слышите о маленьких/средних/больших/модульных/интеграционных/функциональных/сценарных тестах, но сколько из нас знают что это означает? В данной статье мой взгляд на виды тестов.

Модульные/Маленькие

Начнем с модульных тестов. Лучшее определение, которое я нашел — это тесты, которые выполняются очень быстро (менее 1 миллисекунды) и в случае, когда они не проходят, нам не нужен дебагер для выявления места, где произошла ошибка. Само определение делает ограничение на то, какими должны быть тесты. Например, ваши тесты не должны выполнять никаких операций ввода/вывода (I/O) — это одно из условий выполнения теста менее 1 миллисекунды. Ограничение в 1 миллисекунду очень важно, поскольку вы желаете запускать ВСЕ (тысячи) ваши модульные тесты каждый раз, когда вы изменяете что-либо, желательно [при каждом сохранении файла](http://habrahabr.ru/blogs/testing/64737/). Я могу терпеть только 2 секунды и в течение этих двух секунд я хочу, чтобы все мои тесты выполнились и ничего не сломалось. Это очень здорово, когда можно просто нажать ctrl+z несколько раз для отмены недавних изменений и исправить «красный» тест. Немедленная обратная связь вызывает привычку как у наркомана. Отсутствие потребности в дебагере подразумевает, что область теста локализована (отсюда и название теста — модульный (unit), тест одного класса, например).  
  
Цель модульных тестов проверить условную логику в вашем коде — ваши if'ы и циклы. Это те места, где рождаются большинство багов (смотри [теорию багов](http://misko.hevery.com/2008/11/17/unified-theory-of-bugs/)). Вот почему, в случае если вы не делаете другого тестирования, модульные тесты лучшее средство, чтобы выявить простые ошибки в самом начале. Если вы имеете код пропущенный через модульные тесты, то вам будет проще тестировать на более высоких уровнях, описанных ниже.  
  
[KeyedMultiStackTest.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/test/java/com/google/test/metric/collection/KeyedMultiStackTest.java) — это пример хорошего модульного теста из [Обозревателя тестируемости](http://code.google.com/p/testability-explorer). Почувствуйте, как каждый тест рассказывает нам историю. Это не testMethodA, testMethodB,... — это скорее похоже на сценарий. Заметьте, как вначале идут обычные тесты, к которым вы возможно привыкли, но ближе к концу файла тесты выглядят немного незнакомо. Это объясняется тем, что это тесты на пограничные случаи, которые я обнаружил уже позже. Забавная вещь с тестируемым классом в [KeyMultiStack.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/main/java/com/google/test/metric/collection/KeyedMultiStack.java) в том, что я переписывал его три раза. Я не мог заставить его работать на всех тестах. Один из тестов всегда ломался до тех пор, пока я не понял, что алгоритм содержит существенный недостаток. К тому времени у меня была почти работающая программа и это был ключевой класс для процесса анализа байт-кода. Как вы будете себя чувствовать, когда надо похоронить что-то столь фундаментальное в системе и переписать это с нуля? У меня ушло два дня на переработку до того момента, пока успешно не прошли все тесты. После этого все приложение осталось работоспособным. Это то, что называется — «Ага! момент» — момент, когда вы осознаете насколько хороши модульные тесты.  
  
Должен ли каждый класс иметь соответствующий модульный тест? В общем-то, нет. Многие классы тестируются не напрямую, а через тестирование чего-то еще. Обычно простые объекты-значения (value objects) не имеет модульных тестов. Но не путайте неимение тестов и неполное покрытие тестами. Все классы/методы должны быть протестированы (test coverage). Если вы тест-инфицированны (TDD), то это у вас в крови.

Средние/Функциональные

Итак, с помощью модульных тестов вы удостоверились, что каждый класс по отдельности выполняет то, что нужно. Но как вам узнать, что они также хорошо будут работать вместе? Для этого нам нужно объединить связанные классы так, как это будет в рабочей версии программы и проверить основные пути их взаимодействия. Задача состоит не в проверке работают ли if'ы или циклы, а в контроле контрактов между интерфейсами классов. Хорошим примером функционального теста является [MetricComputerTest.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/test/java/com/google/test/metric/MetricComputerTest.java). Заметьте, что входными данными для каждого теста является внутренний класс в тестируемом файле и выходом является ClassConst.java. Для получения выхода несколько классов взаимодействуют между собой: парсинг байт-кода, анализ путей выполнения и вычисление стоимости выполнения до того момента пока окончательная стоимость не будет принята.  
  
Многие из классов тестируются дважды. Один раз с помощью модульных тестов, описанных выше, и второй раз через функциональные тесты. Если вы уберете модульные тесты, то я все равно глубоко уверен, что функциональные тесты обнаружат те изменения, которые поломали код, но я не буду уверен в какую часть кода надо будет залезть, чтобы исправить поломку, поскольку это может быть в любом классе, задействованном в тесте. Когда функциональный тест не проходит, а модульные тесты выполняются успешно, то мне порой приходится вооружаться дебагером. После того как проблемное место будет обнаружено, я добавляю модульный тест, которой поможет мне 1) доказать, что я понимаю, в чем заключается баг и 2) предотвратить повторное появление бага. Подобные модульные тесты объясняют появление странных тестов в конце [KeyedMultiStackTest.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/test/java/com/google/test/metric/collection/KeyedMultiStackTest.java). Это вещи, о которых я не задумывался при первоначальном написании тестов, но которые необходимо было включить после многих часов отладки и обнаружения источника ошибок в классе из [KeyedMultiStack.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/main/java/com/google/test/metric/collection/KeyedMultiStack.java).  
  
Вычисление метрик является лишь небольшой частью того, что делает Обозреватель тестируемости. Существуют еще функции, которые следует протестировать. Вы можете думать о функциональных тестах, как о тестах связанных классов, которые формируют единую сущность для всего приложения. Вот некоторые из подобных сущностей в Обозревателе: парсинг байт-кода java, парсинг кода java, парсинг кода с++, высисление метрик, 3 разных вида отчетов и движок для советов. Все они соответствуют уникальному набору классов, которые взаимодействуют между собой и нуждаются в совместном тестировании. Чаще всего подобные наборы не зависят друг от друга.

Большие/Приемочные/Сценарные

Что мы еще должны протестировать, кроме if'ов, циклов и взаимодействия интерфейсов. Существует еще один класс возможных ошибок. Вы можете неверно объединить компоненты системы. Например, можно передать null вместо отчета, неправильно указать путь к jar-файлу для парсинга и т.д. Это не логические баги, а баги связывания. К счастью, баги связывания легко воспроизводимы и обычно сопровождаются появлением exception'а. Примером приемочного теста является [TestabilityRunnerTest.java](http://code.google.com/p/testability-explorer/source/browse/trunk/core/src/test/java/com/google/test/metric/TestabilityRunnerTest.java). Посмотрите как тесты проверяют работу всего приложения и имеют не так уж много assert-проверок. Нам их много и не нужно — мы до этого уже убедились, что все функции работают и нам надо только проверить те немногие места связки компонентов.