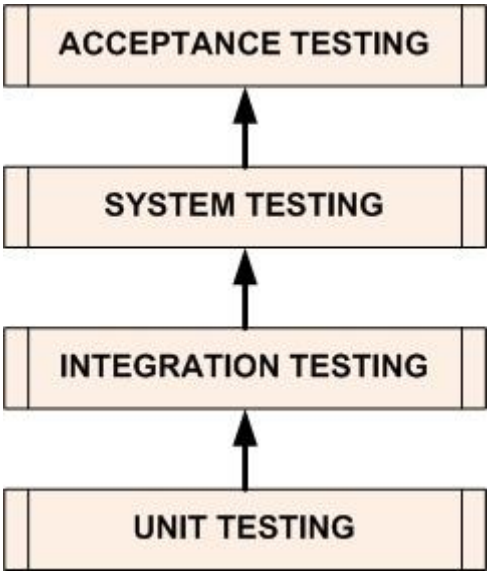
Содержание

Уровни тестирования

В Тестировании ПО можно выделить 4 типичных уровня тестирования:



- **Модульное Тестирование (Unit Testing)** — модуль это наименьшая функциональная часть программы или приложения, которая не может функционировать отдельно, а только лишь в сочетании с другими модулями. Тем не менее, после разработки этого модуля мы уже можем приступить к тестированию и найти несоответствия с нашими требованиями. Модульное тестирование заключается в тестировании этого отдельного модуля, как части программы, подразумевая, что это только модуль и он не может существовать самостоятельно и является частью приложения, программы
- **Интеграционное Тестирование (Integration Testing)** — следующий уровень тестирования, который проводится после модульного тестирования. После того как отдельные модули нашего приложения были протестированы, нам следует провести интеграционное тестирование, чтобы убедиться что наши модули успешно функционируют в связке друг с другом. Иными словами тестируем 2 и более связанных модуля, чтобы проверить что интеграция прошла успешно и без явных багов.

- **Системное Тестирование (System Testing)** — уровень тестирования, в котором мы проводим тестирование целой системы или приложения, полностью разработанного и которое уже готово к потенциальному релизу. На этом уровне мы тестируем систему, приложение в целом, проводим тестирования на всех требуемых браузерах или операционных системах (если десктоп приложение) и проводим все требуемые типы тестирования такие как: функциональное, тестирование безопасности, тестирование юзабилити, тестирование производительности, нагрузочное тестирование и т.д.
- **Приемочное Тестирование (Acceptance Testing)**— после успешного завершения системного тестирования, продукт проходит уровень приемочного тестирования, который обычно проводится заказчиком или любыми другими заинтересованными лицами, с целью убеждения, что продукт выглядит и работает так, как требовалось изначально и было описано в требованиях к продукту. Приемочное тестирование также может проводиться после каждого из вышеописанных уровней тестирования

Unit-тестирование

Юнит-тесты (вики) — это тестирование одного модуля кода (обычно это одна функция или один класс в случае ООП-кода) в изолированном окружении. Это значит, что если код использует какие-то сторонние классы, то вместо них подсовываются классы-заглушки (моки и стабы), код не должен работать с сетью (и внешними серверами), файлами, базой данных (иначе мы тестируем не саму функцию или класс, а еще и диск, базу, и т.д.).

Обычно юнит-тест передает функции разные входные данные и проверяет, что она вернет ожидаемый результат. Например, если у нас есть функция проверки правильности номера телефона, мы даем ей заранее подготовленные номера, и проверяем, что она определит их правильно. Если у нас есть функция решения квадратного уравнения, мы проверяем, что она возвращает правильные корни (для этого мы заранее делаем список уравнений с ответами).

Юнит-тесты хорошо тестируют такой код, который содержит какую-то логику. Если в коде мало логики, а в основном содержатся обращения к другим классам, то юнит-тесты написать может быть сложно (так как надо сделать замену для других классов и не очень понятно, что именно проверять?).

За долгие годы существования понятия unit-тестирования были выработаны некоторые подходы, которые помогают несколько автоматизировать создание тестов. «Автоматизировать» в данном случае не означает, что за вас какая-то программа напишет тесты. Она означает, что есть определенная стратегия написания, которая упрощает работы над созданием тестов.

Некоторые тесты можно запустить и проверить программно, некоторые (особенно это касается UI) — крайне сложно. Для того же UI бывает удобно просто записать движения мышки, нажатие клавиатуры и потом эти действия «натравить» на вашу систему. И там отслеживать, что и как обрабатывает такое поведение.

Каждый раз при запуске тест должен пройти через 4 стадии:

1. Инициализация — надо подготовить необходимые данные для теста

2. Запуск тестируемой функции/метода — т.е. мы просто выполняем проверяемую функцию
3. Проверка — на этом этапе нам надо проверить результат выполнения
4. Освобождение ресурсов — надо «почистить» наши изменения

Инициализация

На самом деле это один из сложных этапов. Надо создать для программы полную иллюзию того, что она работает в обычном режиме. А для того же сервлета такое сделать не так уж и просто.

Или работа с базой данных — для заполнения какой-либо таблицы нужен заранее набор справочных данных. Тех же контрагентов, пользователей, прав и прочая. Откуда это все появится? Надо будет что-то придумывать.

Да и нередко бывает, что для метода какого-то класса надо создать еще десяток объектов. В общем, задачка достаточно сложная. Иногда можно пойти по пути создания окружения для каждого теста в отдельности, иногда — для всех тестов вместе.

Рекомендуется обратить внимание на целую коллекцию так называемых Mock-объектов. Для примера — вы хотите проверить свой сервлет. Но на вход ему надо подать объекты, которые реализуют интерфейсы HttpServletRequest b YttpServletResponse. Web-сервер их предоставляет, но без него их надо как-то создавать самим. Так вот есть такие уже готовые. Просто создаете их, добавляете нужные параметры и используете.

Запуск

Здесь особо говорить не о чем — запуск он и в Африке запуск. Надо выполнить метод или несколько. Если вы все необходимые переменные, данные создали на этапе инициализации, то здесь особо и говорить не о чем. Правда есть момент — тесты должны выполняться быстро. Вряд ли вам понравиться, что при запуске тестов можно идти пить кофе каждый раз. При такой производительности вас и с работы могут попросить. Поэтому иногда приходится придумывать методы, которые позволяют выполнять методы быстро. Часто это связано с обращениям к внешним ресурсам — базы данных, внешние службы. Иногда бывает так, что надо обратиться к тому, чего вообще нет — например вам надо работать с кассовым аппаратом. Приходится пользоваться разделением интерфейсов и реализаций. Ваша система работает с интерфейсом, за которым «прячется» эмулятор кассового аппарата. Так что если вдруг у вас нет реального объекта для тестирования — заменяйте его фиктивным и тестируйте свою систему с таким. Что же касается базы данных — иногда тестировать надо не саму работу с базой данных, а обработку данных из нее. Можно сделать опять же какой-нибудь заменитель, который будет возвращать нужные нам данные. А это будет быстрее, чем к рельной базе ходить. Особенно если учесть, что таких обращений может быть много.

Здесь можно посоветовать следующее — проектируйте сразу с учетом того, что надо будет писать тесты.

Проверка

Здесь надо просто проверить, что система сделал то, что от нее и требовалось. Либо проверив какие-то данные, либо как-то отследив действия системы (по логам, по вызовам — заменить вызовы реальные на вызовы подставных объектов). По данным конечно

проще, да и часто этого хватает. Отслеживание действия — задача более серьезная. Здесь надо предусмотреть обработку разных веток программы, верную реакцию на исключения. А если учесть, что такие исключения бывают не только внутри программы, но и снаружи, от внешних ресурсов — задачка нетривиальная.

Освобождение ресурсов

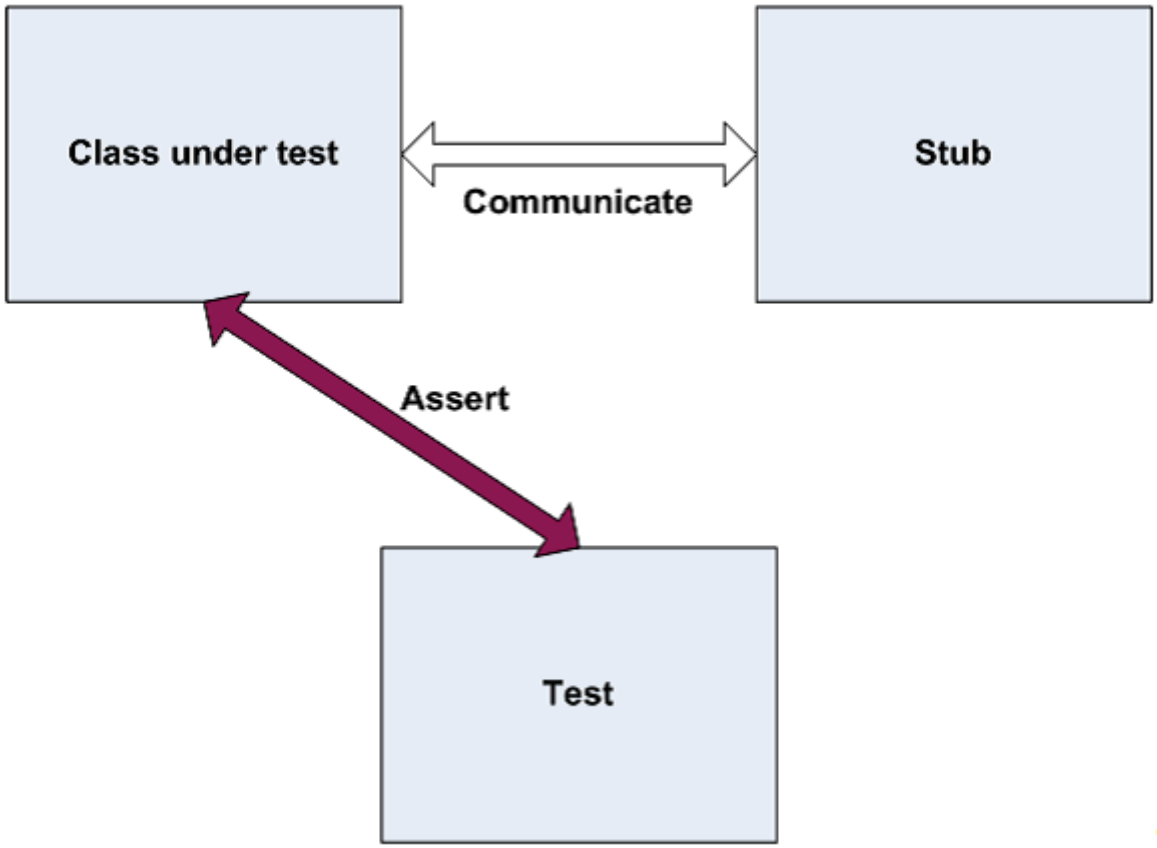
Здесь надо просто сделать все, чтобы ваши действия вернули систему в состояние до ваших разрушительных действий. Удаляйте данные, освобождайте ресурсы — в общем сами понимаете, что здесь тоже есть над чем подумать.

Но реализация этих этапов — задача достаточно творческая. Надо копить опыт, придумывать, пробовать. И когда ты начинаешь понимать и принимать идею unit-тестов — они становятся очень интересной областью.

Как тестировать код, если вызов одного метода влечет за собой цепочку вплоть до базы данных? В таких случаях на помощь приходят так называемые фейковые-объекты, предназначенные для симуляции поведения реальных объектов во время тестирования. **Выделяют два типа подделок: стабы (stubs) и моки (mock).** Часто эти понятия путают. Разница в том, что стаб ничего не проверяет, а лишь имитирует заданное состояние. А мок – это объект, у которого есть ожидания. Например, что данный метод класса должен быть вызван определенное число раз. Иными словами, ваш тест никогда не сломается из-за «стаба», а вот из-за мока может.

С технической точки зрения это значит, что используя стабы в Assert мы проверяем состояние тестируемого класса или результат выполненного метода. При использовании мока мы проверяем, соответствуют ли ожидания мока поведению тестируемого класса.

Стабы — это классы-заглушки, которые вместо выполнения действия возвращают какие-то данные (то есть по сути функция состоит из одного return). Например, стаб класса работы с базой данных может вместо реального обращения к базе данных возвращать, что запрос успешно выполнен. А при попытке прочитать что-то из нее возвращает заранее подготовленный массив с данными.



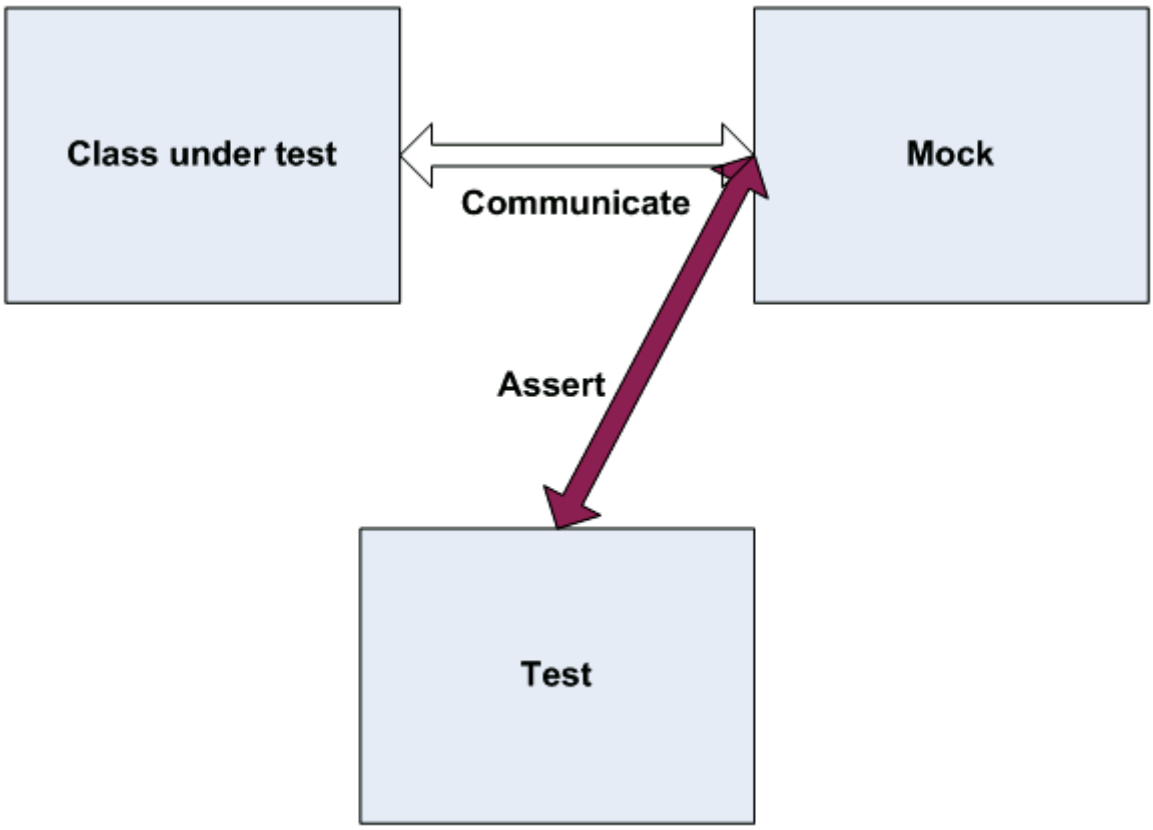

```
[Test]
public void LogIn_ExisingUser_HashReturned()
{
        // Arrange
        OrderProcessor = Mock.Of<IOrderProcessor>();
        OrderData = Mock.Of<IOrderData>();
        LayoutManager = Mock.Of<ILayoutManager>();
        NewsProvider = Mock.Of<INewsProvider>();

        Service = new IosService(
                UserManager,
                AccountData,
                OrderProcessor,
                OrderData,
                LayoutManager,
                NewsProvider);

        // Act
        var hash = Service.LogIn("ValidUser", "Password");

        // Assert
        Assert.That(!string.IsNullOrEmpty(hash));
}
```

Моки — это классы-заглушки, которые используются чтобы проверить, что определенная функция была вызвана.




```
[Test]
public void Create_AddAccountToSpecificUser_AccountCreatedAndAddedToUser()
{
    // Arrange
    var account = Mock.Of<AccountViewModel>();

    // Act
    _controller.Create(1, account);

    // Assert
    _accountData.Verify(m => m.CreateAccount(It.IsAny<IAccount>()), Times.Exactly(1));
    _accountData.Verify(m => m.AddAccountToUser(It.IsAny<int>(), It.IsAny<int>()), Times.Once());
}
```

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.

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

Тесты должны базироваться на спецификации.

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.

Организация тестирования

Тесты (test cases) могут группироваться в наборы (test suites). В случае провала тестирования, программист увидит имена виновников – наборов и тестов. В связи с этим, имя должно сообщать программисту о причинах неприятности и вовлеченных модулях.

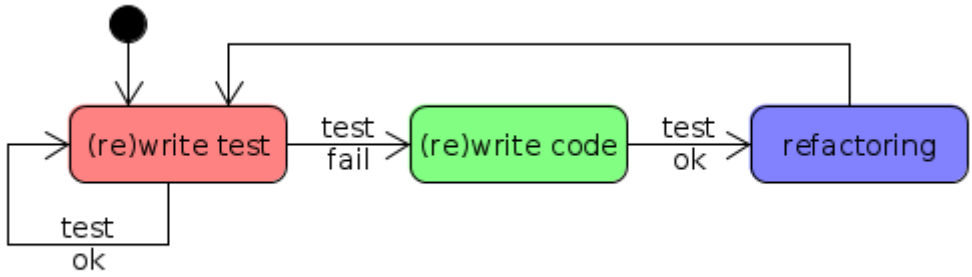
Имя может быть полезным и в других случаях. Например, заказчик нашел в программе ошибку, значит программист должен либо добавить новый тест, либо исправить существующий. Следовательно, нужна возможность найти тест по характеру ошибки, а помочь в этом могут хорошо выбранные имена.

Обычно имя теста состоит из наименования тестируемого класса или функции и отражает особенности проверяемого поведения. Например, имена TestIncorrectLogin и TestWeakPasswordRegistration могли бы использоваться для проверки корректности логина и реакции модуля на слишком простой пароль.

Тесты могут располагаться как вместе с тестируемым кодом, так и в отдельном проекте. Очевидно, что основной проект не должен зависеть от тестов, ему незачем знать что его кто-то тестирует. Разделение тестов и рабочего кода считается хорошим тоном, однако бывают исключения – например, если нам надо проверить реализацию, т.е. корректность работы его закрытых (private) методов класса. В таком случае тесты могут размещаться прямо внутри тестируемого кода (или используется отношение дружбы между тестируемым классом и тестом). Обычно такой необходимости не возникает, а тестами покрывают лишь интерфейс, т.к. именно он формирует поведение класса.

Мало того, что основной проект не должен зависеть от тестов – тесты не должны зависеть друг от друга. В противном случае на результаты тестирования может оказывать влияние порядок выполнения тестов (т.е. фаза луны). Из этого простого правила можно сделать следующие важные выводы:

- недопустимо изменять глобальные объекты в тестах и тестируемых модулях. Если же тестируемый код взаимодействует с глобальным объектом (например базой данных), то проверка корректности такого взаимодействия – удел системного тестирования;
- каждый тестовый случай должен быть настроен на выявление лишь одной возможной ошибки – иначе по имени ошибки нельзя установить причину неполадок;
- модули проекта должны соблюдать принцип единой ответственности (Single responsibility principle, SRP) – в противном случае наши тесты будут выявлять более чем по одной ошибке;
- тесты не должны дублировать друг друга. Методология разработки через тестирование (test driven developing, TDD) предполагает короткие итерации, каждая из которых содержит этап рефакторинга (переработки кода). Переработке должен подвергаться не только код основного проекта, но и тесты.



Наконец, юнит-тесты запускаются достаточно часто, поэтому должны работать быстро. В тестовых случаях не должны обрабатываться большие объемы информации – тестовые случаи должны содержать только то, что связано непосредственно с проверяемым поведением. Проверка корректности работы системы на больших объемах данных должна быть вынесена на этап нагрузочного тестирования.

Шаблон написание unit-теста

Использование шаблона Arrange-Act-Assert (AAA) при написании unit тестов, намного повышает шансы других разработчиков понять ваш код. Наверняка вы уже использовали AAA в своих проектах, но не догадывались об этом. Давайте разберемся, что же это за шаблон. Итак, все просто, данный шаблон всего лишь разделяет и группирует код теста на 3 секции, придавая, удобную для чтения, структуру unit теста:
Arrange - выставление начальных условий.
Act - отработка тестируемого функционала.
Assert - сверка ожидаемых значений с полученными.

Рассмотрим простой пример:

```
// Переводчик.
interface ITranslator
{
   string Translate(string line);
}

class EngRusTranslator : ITranslator
{
   public EngRusTranslator(/*Требуем словарь*/)
   {
   }

   public string Translate(string line)
   {
      // Здесь у нас есть доступ к словарю
      // (который надо передавать в
      // качестве аргумента в конструктор),
      // по которому происходит перевод.
      //
      // Но предположим, что он у нас зашит к коде.

      throw new NotImplementedException();
   }
}

[Test]
public void TestTranslate()
{
   // Arrange.
   //
   // Здесь мы можем setup-ить наши Mock объекты,
   // например создать Mock-словарь, который потом
   // передать в качестве аргумента в конструктор.
   ITranslator translator = new EngRusTranslator();

   // Act.
   //
```

```
    // Отработка тестируемого функционала.
    string result = translator.Translate("Hello, World!");

    // Assert.
    //
    // Проверка. Assert или Verify метод.
    Assert.AreEqual("Привет, Мир!", result);
}
```

Хоть пример и простой, но мы явно видим плюсы разработки при таком подходе:

- Assert-методы никогда не перемешаются с Act-методами.
- Неявное навязывание писать ОДИН Assert на ОДИН тест.
- Упрощенный рефакторинг, вам легко будет обнаружить Arrange-блоки, которые можно вынести в SetUp-метод.

Преимущества Unit-тестирования

Тесты как документация

Юнит-тесты могут служить в качестве документации к коду. Грамотный набор тестов, который покрывает возможные способы использования, ограничения и потенциальные ошибки, ничуть не хуже специально написанных примеров, и, кроме того, его можно скомпилировать и убедиться в корректности реализации.

Упрощение архитектуры приложения

Основное правило здесь формулируется следующим образом: «Реализуется только то, что действительно нужно». Если в тесте описаны все сценарии и больше не удаётся придумать, как сломать логику, то нужно остановиться, привести код в более-менее приличный вид (выполнить рефакторинг) и со спокойной совестью лечь спать.

Возможность лучше разобраться в коде

Когда вы разбираетесь в плохо документированном сложном старом коде, попробуйте написать для него тесты. Это может быть непросто, но достаточно полезно, так как:

- они позволят вам убедиться, что вы правильно понимаете, как работает код;
- они будут служить документацией для тех, кто будет читать код после вас;
- если вы планируете рефакторинг, тесты помогут вам убедиться в корректности изменений.

Тестовое покрытие

Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода. Если рассматривать тестирование как "проверку соответствия между реальным и

ожидаемым поведением программы, осуществляемая на конечном наборе тестов", то именно этот конечный набор тестов и будет определять тестовое покрытие:
Чем выше требуемый уровень тестового покрытия, тем больше тестов будет выбрано, для проверки тестируемых требований или исполняемого кода.

Сложность современного программного обеспечения и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более-менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.

Существуют следующие подходы к оценке и измерению тестового покрытия:

1. **Покрытие требований (Requirements Coverage)** - оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).
2. **Покрытие кода (Code Coverage)** - оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.
3. **Тестовое покрытие на базе анализа потока управления** - оценка покрытия основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

Расчет тестового покрытия относительно исполняемого кода программного обеспечения проводится по формуле:
Tcov = (Ltc/Lcode) * 100%, где:
Tcov - тестовое покрытие
Ltc - кол-ва строк кода, покрытых тестами
Lcode - общее кол-во строк кода.

В настоящее время существует инструментарий (например: **Clover**), позволяющий проанализировать в какие строки были вхождения во время проведения тестирования, благодаря чему можно значительно увеличить покрытие, добавив новые тесты для конкретных случаев, а также избавиться от дублирующих тестов. Проведение такого анализа кода и последующая оптимизация покрытия достаточно легко реализуется в рамках тестирования белого ящика (white-box testing) при модульном, интеграционном и системном тестировании; при тестировании же черного ящика (black-box testing) задача становится довольно дорогостоящей, так как требует много времени и ресурсов на установку, конфигурацию и анализ результатов работы, как со стороны тестировщиков, так и разработчиков.

Покрытие кода является важной метрикой для обеспечения качества тестируемого приложения, особенно если речь о проектах со сложной логикой и большим объемом кода. Анализ покрытия кода выполняется с помощью специального инструментария, который позволяет проследить в какие строки, ветви и т.д. кода, были вхождения во время работы автотестов. Наиболее известные инструменты для проведения измерения покрытия кода: AQTime, Bounds Checker, Bullseye Coverage, Coverage Meter, Clover, NCover, IBM Rational PurifyPlus, Intel Compiler, Intel Code Coverage Tool Prototype, JetBrains. С помощью анализа покрытия кода можно оценить плотность покрытия авто-тестами исполняемого кода тестируемого приложения (можно ответить на вопрос «какой объем тестирования мы (наши автотесты) выполняем?»). При детальном анализе

результатов покрытия кода автотестами можно оценить покрытие отдельных компонентов системы (т.е. можно ответить на вопросы: что и в каком объеме мы тестируем?, в каких местах нужно оптимизировать покрытие?, какие места системы не проверяются тестами? и т.д.). Таким образом, зная данную метрику, станет ясно для каких тестовых случаев нужно создать новые тесты, или убрать дублирующие тесты. Данные мероприятия помогут увеличить значение метрики Code Coverage, что в свою очередь должно повысить качество кода и качество тестируемого приложения в целом. Естественно, чем выше показатель данной метрики — тем лучше, однако уже хорошо, если у вас покрыты тестами наиболее сложные и важные фрагменты кода.

Различают несколько способов измерения покрытия кода. В Википедии представлены следующие определения:

- **Покрытие операторов** — каждая ли строка исходного кода была выполнена и протестирована?;
- **Покрытие условий** — каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована?;
- **Покрытие путей** — все ли возможные пути через заданную часть кода были выполнены и протестированы?;
- **Покрытие функций** — каждая ли функция программы была выполнена?;
- **Покрытие вход/выход** — все ли вызовы функций и возвраты из них были выполнены
- **Покрытие комбинаций** — проверка всех возможных комбинаций результатов условий.

В глоссарии ISTQB предоставлен более полный список способов измерения покрытия кода. Однако общий смысл идентичен, но желающим все же рекомендую туда заглянуть,

Примеры написания модульных тестов в Visual Studio приведены здесь:
https://msdn.microsoft.com/en-us/library/dd286656(v=vs.100).aspx
https://msdn.microsoft.com/ru-ru/library/ms182532.aspx

Контрольные вопросы:

1. В чем отличие приемочного тестирования от системного?
2. Что такое моки и стабы?
3. Как осуществляется планирование тестов?
4. Какие правила организации тестов вы знаете?
5. Что за шаблон Arrange-Act-Assert?
6. Какие преимущества применения unit-тестирования?
7. Что такое тестовое покрытие?

Ссылки:

1. http://software-testing.org/testing/urovni-testirovaniya-testing-levels-v-testirovanii-po.html
2. https://tproger.ru/translations/unit-tests-purposes/
3. https://habrahabr.ru/post/191986/
4. http://losev-al.blogspot.ru/2013/01/unit-testing.html
5. http://java-course.ru/student/book1/unit-test/
6. http://citforum.ru/SE/testing/unit_testing/
7. http://pro-prof.com/archives/1549
8. https://habrahabr.ru/post/134836/
9. http://merle-amber.blogspot.ru/2008/09/mock.html
10. http://www.javaworld.com/article/2074508/core-java/mocks-and-stubs---understanding-test-doubles-with-mockito.html
11. http://itvdn.com/ru/video/unit-testing-csharp
12. http://www.handcode.ru/2010/04/arrange-act-assert.html