**Тестирование с точки зрения разработчика**

Класс-слушатель **CalcListener**, который будет дополнительно использоваться в итоговом примере, (как я понял) необходим для укороченного вывода сообщений при тестировании.

Существует целая область знаний посвященная тестам. Тесты типа «черного ящика», тесты типа «белого ящика», стратегии тестирования и прочая.  
Но зачем они разработчику ? Он же написал свою функцию, проверил, что она работает. И пусть тестер мучается, чтобы приложение все-таки свалилось. Это его работа — находить баги в супер-проверенном коде. А мы уж исправим. Работа такая.  
Ответ очень простой — для того, чтобы убедиться, что весь предыдущий код работает также как и до наших изменений. Чем больше мы можем проверить без помощи человека — тем лучше. Основные вопросы, которые возникают в процессе придумывания тестов — что же надо тестировать и как. Иногда доходит до абсурда — ты видишь, что вставил запись — считывать ее опять? Так ведь это же очевидно — раз вставилась запись — само собой получим тоже самое в ответе. Или зачем проверять выборку если она работает ? Или может проверить, что корректно работают set и get? Я некоторое время (правда это было давно) здорово мучался с этими вопросами. Полагаю, что я был не один. Единого подхода КАК писать — не выработал. Но ЧТО надо тестировать — у меня появился простой рецепт: **Тестируйте то, что вызывает функции, которые могу измениться.** Исходя из этого и придумывайте тесты. Давайте рассмотрим это на примере.  
Итак у нас есть таблица студентов, для которой у нас есть методы модификации. Представим, что разработчики получили указание, что поле «Специальность» надо переименовать. Придется нам изменить методы считывания данных, вставки, модификации и удаления. Быстро переименовав поле в таблице и во всех запросах мы проверяем, что наши методы работают корректно и рапортуем о завершении ? Конечно не так — нам надо также проверить все места, где вызываются наши методы и проверить там корректность передаваемых параметров. Хорошо, если только одно приложение использует наш замечательный класс ManagementSystem. А если их несколько ? Все будем запускать и все будем проверять ? А если их хотя бы 3 и в каждом вызовов по 10-15 ? И надо все помнить. И это еще очень небольшой проект. Что же происходит, когда в проекте миллион строк ? Ощутили проблемы модификации существующегго кода ? Я когда-то это сильно ощутил. Мне предложили ввести дополнительный параметр (транзакцию) в проекте из 50 Мб исходных кодов. Тестов там не было. Провозился я тогда более 2 недель. Только над серверной частью. А там была еще часть из JSP. И я не был уверен, что все работает корректно. Потому что одних таблиц было порядка 200. И каждое обращение к таблицам надо было обернуть транзакцией и проверить, что все работает корректно.

Так какова стратегия написания тестов для нашего случая ? Т.к. база может меняться, то мы должны иметь быстрый способ проверить — все ли методы, которые работают непосредственно с базой, в порядке. Можно написать простые методы, которые что-то модифицируют в базе и хотя бы не вызывают ошибки выполнения. Уже прогресс. Добавили поле и запустили сразу тесты — получили список мест, где надо исправлять. Можно сделать глупый на первый взгляд тест — записали в базу, считали и проверили с тем, что раньше записывали. Да что там может случится? Оказывается может. Триггер разработчики базы ввели. А мы не знали. И получаем не то, что рассчитывали. Теперь либо меняем тест, либо меняем код.  
И вот наш глупый тест, который вставляет данные, а потом их же считывает, оказывается очень нужным и удобным. Во-первых мы проверим, что это просто работает. Во-вторых — мы убедимся в том, что данные записываются именно так, как нам надо. В-третьих — для этого мы не запускаем приложение и не работаем с ним сами.  
Мы можем это делать сразу после сборки тем же самым Ant’ом — создадим цель «test» и запустим класс, который будет вызывать все, что только можно протестировать.  
Заметьте — не ищем по коду, не запускаем приложение, не смотрим в базу с целью понять, что же вставилось. Мы просто запускаем много тестов и смотрим результат. Очень мощный способ повышения эффективности разработки. Даже если Вы были единственным разработчиком, то спустя 3-4 месяца Вы вряд ли вспомните, что же надо исправлять точно. А тут Вам все покажут Вами же написанные тесты. Повторюсь еще раз — тесты пишутся не для того, чтобы найти ошибку (это задача тестеров), а только для того, чтобы проверить, то что Вы только что написали. И что самое главное — они пишутся для проверки корректности будущих изменений. Если Вы будете исходить из этой стратегии, то писать и придумывать тесты будет проще — Вам нужно придумать тест, который может автоматически проверить, работает ли Ваша функция вообще и корректно ли она все это делает. И если Вы будете еще понимать что может измениться, то и тесты можно будет писать еще более корректно.

Стадии тестирования

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. Ну 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. Выбирать Вам.  
Очень советую 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 и прочая создали на этапи инициализации, то здесь особо и говорить не о чем. Правда есть момент, оторый я бы выделил — 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, котоыре 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 вооще 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 тестироваиня — 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-нибуль заменитель, который будет возвращать нужные нам данные. А это будет быстрее, чем к рельной базе ходить. Особенно если учесть, что такх 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**  
Ну здесь надо просто сделать все, чтобы ваши действия вернули систему в сосотояние 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. Так что получайте удовольствие

А теперь мы расмотрим один из самых известных продуктов для создания Unit-тестов — JUnit

JUnit — один из самых распространенных framework для тестирования

По большому счету JUnit прямо-таки издевательски простой пакет. Вся его задача состоит в том, чтобы удобно запустить некий класс, который будет состоять из функций предназначенных для тестирования Вашего приложения. В нем все предназначено только для одной цели — удобно писать и запускать тесты, которые Вы придумали сами. В общем-то и все. Если этот пакет освоить хорошо, то он становиться так же необохдим и удобен как тот же FAR или TotalCommander.

Мы рассмотрим JUnit на примере сравнительно новой версии 4.4 (со временем она тоже будет устаревшей). Загрузить ее можно с сайта —<http://www.junit.org/> Наверху справа есть пункт «Download JUnit». Дальше Вас кинут на sourceforge.net  
<http://sourceforge.net/project/showfiles.php?group_id=15278&package_id=12472>  
В процессе изучения мы будем вводить классы, которые используются в JUnit — после распаковки Вы можете найти на них документацию в каталоге doc — там описание API. Я постараюсь остановиться на JUnit подробнее потому, что это очень важный и нужный инструмент. Но, как говаривал Козьма Прутков, нельзя объять необъятное   
Также стоит обратить внимание на то, что версия 4 требует JAVA 1.5 и выше.  
Учтите, что API версии 4 отличается от версии 3.8, причем очень сильно. В данной статье я рассмотрю некоторые возможности, которые делают тестирование более комфортным. Но сказать, что это в разы облегчит Вам написание тестов — я бы так не сказал. Но тем не менее — если Вы привыкните писать unit-тесты, то это очень сильно поможет Вам в разработке. Ну а теперь за дело.

**Первый простой тест**  
Давайте создадим несложный класс, который будет делать очень простую операцию — складывать числа.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Calc  {    public int getSum(int x, int y)    {      return x+y;    }  } |

Теперь мы должны написать класс, который тестирует наше «сложное» приложение. В отличии от версии 3.8, где тестирующий класс должен был наследоваться от класса TestCase и все методы для тестирования должны были начинаться с «test» (Например: testSum, testCalculation и т.д.), теперь этого всего не надо. Все необходимое нам заменяет аннотация — об этом Вы можете прочесть в статье [Java 5.0″](http://forum.vingrad.ru/articles/topic-157946.html). Наш класс выглядит очень просто

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import org.junit.Test;  // Еще одна "вкусность" Java 5 - можно импортировать static-методы  import static org.junit.Assert.assertEquals;    public class TestCalc {      // @Test - это аннотация, которая обозначает,      // что метод должен быть вызван для тестирования чего-нибудь      @Test      public void getSumTest() {          Calc c = new Calc();          // Этот метод вызовет исключение, если результат нашего калькулятора будет отличен от 50          assertEquals(c.getSum(20, 30), 50);      }  } |

Положите в каталог с нашими классами файл junit-4.4.jar (Который Вы должны скачать). Для сборки нашего приложения запускаем команду:  
javac -cp junit-4.4.jar \*.java

Теперь запускаем  
java -cp .;junit-4.4.jar -ea org.junit.runner.JUnitCore TestCalc

Ключик (-ea) разрешает обработку assert — в общем-то особо нам это не важно, но как-то привычнее. И … на экране много не видно

|  |  |
| --- | --- |
| 1  2  3  4  5 | JUnit version 4.4  .  Time: 0,031    OK (1 test) |

Это все. Информации в общем-то немного и запускать тоже пока не очень удобно. Количество точек после строки JUnit version 4.4 указывает, сколько тестов мы прошли. Как Вы возможно догадались все тесты исполняет специальный «запускатель» JUnitCore. Ему мы передаем класс, который будем тестировать.  
Понятно, что для тестирования одного метода мы написали больше, чем нужно. Целый класс. Да еще и «запускатель». Не много ли чести   
Но заметьте одну ГЛАВНУЮ ОСОБЕННОСТЬ — МЫ НИГДЕ не вызываем метод getSumTest. За нас его исполняет JUnitCore. Он находит методы, которые мы обозначили аннотацией @Test и выполняет их. Т.е. если мы напишем еще 10 методов для Calc, то для тестирования в нашем классе TestCalc мы напишем еще 10 методов (может и больше — ведь тестировать метод можно несколькими способами). Но мы НИГДЕ не будем писать код для ВЫЗОВА этих методов. Правда пока мы не умеем управлять порядком наших тестов, но это возможно и мы сможем посмотреть как это делается.  
Давайте «испортим» наш класс Calc. По каким-то внутренним требованиям кто-то переписал наш замечательный метод так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Calc  {    public int getSum(int x, int y)    {      return x+y;    }      public int getSubtraction(int x, int y)    {      return x-y-1;    }  } |

Как видите наш второй метод содержит ошибку — разность в общем-то не так вычисляется. Класс TestCalc мы тоже должны переписать. Точнее дополнить еще одним методом для тестирования вычитания. Но тестирует он «верную» версию — т.е. вычитает по правилам арифметики.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import org.junit.Test;  import static org.junit.Assert.assertEquals;    public class TestCalc {      // @Test - это аннотация, которая обозначает, что метод должен быть вызван для тестирования чего-нибудь        @Test      public void getSumTest() {          Calc c = new Calc();          // Этот метод вызовет исключение, если результат нашего калькулятора будет отличен от 50          assertEquals(c.getSum(20, 30), 50);      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          // Этот метод вызовет исключение, если результат нашего калькулятора будет отличен от -10          assertEquals(-10, c.getSubtraction(20, 30));      }  } |

Теперь перекомпилируем наши классы  
javac -cp junit-4.4.jar \*.java

Опять запускаем командой  
java -cp .;junit-4.4.jar -ea org.junit.runner.JUnitCore TestCalc

И видим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | JUnit version 4.4  ..E  Time: 0,047  There was 1 failure:  1) getSubtractionTest(TestCalc)  java.lang.AssertionError: expected:<-10> but was:<-11>          at org.junit.Assert.fail(Assert.java:74)          at org.junit.Assert.failNotEquals(Assert.java:448)          at org.junit.Assert.assertEquals(Assert.java:102)          at org.junit.Assert.assertEquals(Assert.java:323)          at org.junit.Assert.assertEquals(Assert.java:319)          at TestCalc.getSubtractionTest(TestCalc.java:26)          at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)          at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)          at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)          at java.lang.reflect.Method.invoke(Method.java:585)          at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:59)          at org.junit.internal.runners.MethodRoadie.runTestMethod(MethodRoadie.java:98)          at org.junit.internal.runners.MethodRoadie$2.run(MethodRoadie.java:79)          at org.junit.internal.runners.MethodRoadie.runBeforesThenTestThenAfters(MethodRoadie.java:87)          at org.junit.internal.runners.MethodRoadie.runTest(MethodRoadie.java:77)          at org.junit.internal.runners.MethodRoadie.run(MethodRoadie.java:42)          at org.junit.internal.runners.JUnit4ClassRunner.invokeTestMethod(JUnit4ClassRunner.java:88)          at org.junit.internal.runners.JUnit4ClassRunner.runMethods(JUnit4ClassRunner.java:51)          at org.junit.internal.runners.JUnit4ClassRunner$1.run(JUnit4ClassRunner.java:44)          at org.junit.internal.runners.ClassRoadie.runUnprotected(ClassRoadie.java:27)          at org.junit.internal.runners.ClassRoadie.runProtected(ClassRoadie.java:37)          at org.junit.internal.runners.JUnit4ClassRunner.run(JUnit4ClassRunner.java:42)          at org.junit.internal.runners.CompositeRunner.runChildren(CompositeRunner.java:33)          at org.junit.internal.runners.CompositeRunner.run(CompositeRunner.java:28)          at org.junit.runner.JUnitCore.run(JUnitCore.java:130)          at org.junit.runner.JUnitCore.run(JUnitCore.java:109)          at org.junit.runner.JUnitCore.run(JUnitCore.java:100)          at org.junit.runner.JUnitCore.runMain(JUnitCore.java:81)          at org.junit.runner.JUnitCore.main(JUnitCore.java:44)    FAILURES!!!  Tests run: 2,  Failures: 1 |

Тут уже информации гораздо больше. Видно, что не выполнен метод getSubstractionTest и даже есть подсказка для чисел  
expected:<-10> but was:<-11>

Но пока удобств все равно не так уж много. Мало информации, импользуется какой-то JUnitCore. Попробуем сделать наше приложение более информативным и удобным. Изменим наш тестирующий класс — оснастим его методом main. А то надоело вызывать JUnitCore.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import org.junit.Test;  import static org.junit.Assert.assertEquals;    // Теперь мы не будем вызывать в командной строке JUnitCore  import org.junit.runner.JUnitCore;    public class TestCalc {        @Test      public void getSumTest() {          Calc c = new Calc();          assertEquals(50, c.getSum(20, 30));      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          assertEquals(-10, c.getSubtraction(20, 30));      }        // Вот где у нас будет вызов нашего тестирующего класса      public static void main(String[] args) {          JUnitCore core = new JUnitCore();          core.run(TestCalc.class);      }  } |

Ну теперь компилируем и запускаем  
javac -cp junit-4.4.jar \*.java  
java -cp .;junit-4.4.jar -ea TestCalc  
И … Вообще ничего нет. А как же страшные ошибки ? Они же у нас есть. На всякий случай выполним опять через JUnitCore.  
java -cp .;junit-4.4.jar -ea org.junit.runner.JUnitCore TestCalc  
Да нет, все в порядке — ругается наша программа. Открою небольшую тайну — сам JUnitCore не генерит сообщений. Он использует классы, которые унаследованый от RunListener и вызывает его методы при наступлении определенных событий. Если залезть в исходники, то можно увидеть, что JUnitCore регистрирует TextListener, который является наследником RunListener’а.  
Давайте попробуем написать свой слушатель. Наш тестируемый класс Calc.java мы пока оставим в покое. Итак:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import org.junit.Test;  import static org.junit.Assert.assertEquals;    import org.junit.runner.JUnitCore;    import org.junit.runner.notification.RunListener;  import org.junit.runner.notification.Failure;    public class TestCalc {        @Test      public void getSumTest() {          Calc c = new Calc();          assertEquals(50, c.getSum(20, 30));      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          assertEquals(-10, c.getSubtraction(20, 30));      }        public static void main(String[] args) {          JUnitCore core = new JUnitCore();          // Вот подключение нашего собственного слушателя/листенера          core.addListener(new CalcListener());          core.run(TestCalc.class);      }  }    // А вот его реализация  class CalcListener extends RunListener {      // Пока мы сделали один метод - листенер будет реагировать только когда возникаю проблемы        @Override      public void testFailure(Failure fail) {          System.out.println("Failed:" + fail.getDescription().getDisplayName() + " [" + fail.getMessage() + "]");      }  } |

Если Вы откроете документацию и найдете описание RunListener, то сможете написать свои собственные методы для обработки начала тестирования метода, начала тестирования вообще и т.д. Советую прочитать внимательно. Для окончательного слова о листенерах приведем еще один пример кода.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | import org.junit.Test;  import static org.junit.Assert.assertEquals;    import org.junit.runner.JUnitCore;    import org.junit.runner.Description;  import org.junit.runner.notification.RunListener;  import org.junit.runner.notification.Failure;    public class TestCalc {      // @Test - это аннотация, которая обозначает, что метод должен быть вызван для тестирования чего-нибудь        @Test      public void getSumTest() {          Calc c = new Calc();          assertEquals(50, c.getSum(20, 30));      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          assertEquals(-10, c.getSubtraction(20, 30));      }        public static void main(String[] args) {          JUnitCore core = new JUnitCore();          core.addListener(new CalcListener());          core.run(TestCalc.class);      }  }    class CalcListener extends RunListener {        @Override      public void testStarted(Description desc) {          System.out.println("Started:" + desc.getDisplayName());      }        @Override      public void testFinished(Description desc) {          System.out.println("Finished:" + desc.getDisplayName());      }        @Override      public void testFailure(Failure fail) {          System.out.println("Failed:" + fail.getDescription().getDisplayName() + " [" + fail.getMessage() + "]");      }  } |

Если Вам не требуется отслеживать имена методов, а просто получить известие о том, что что-то пошло не так можно использовать аннотации@Before, @After, @BeforeClass, @AfterClass. Методы, которые снабжены такой аннотацией будут вызываться: в начале каждого метода, в конце каждого метода, в начале тестирования, в конце тестирования.  
Вот для примера код — обратите внимание, что методы с аннотацией @Before и @After не должны быть static. А методы с аннотацией@BeforeClass и @AfterClass — должны быть static.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | import org.junit.Test;  import org.junit.Before;  import org.junit.After;  import org.junit.BeforeClass;  import org.junit.AfterClass;  import org.junit.Ignore;  import static org.junit.Assert.assertEquals;    import org.junit.runner.JUnitCore;    public class TestCalc {      // @Test - это аннотация, которая обозначает, что метод должен быть вызван для тестирования чего-нибудь        @Test      public void getSumTest() {          Calc c = new Calc();          assertEquals(50, c.getSum(20, 30));      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          assertEquals(-10, c.getSubtraction(20, 30));      }        @BeforeClass      public static void allTestsStarted() {          System.out.println("All tests started");      }        @AfterClass      public static void allTestsFinished() {          System.out.println("All tests finished");      }        @Before      public void testStarted() {          System.out.println("Started");      }        @After      public void testFinished() {          System.out.println("Finished");      }        @Test      // Обратите внимание аннотацию - она говорит, что тест будет проигнорирован. Если ее убрать,      // то сообщение появиться      @Ignore      public void testIgnored() {          System.out.println("Ignored test");      }        public static void main(String[] args) {          JUnitCore core = new JUnitCore();          core.run(TestCalc.class);      }  } |

Порядок прежде всего

Бывают случаи, когда порядок вызовов методов важен дл ятестирования. В таком случае можно воспользоваться сортировкой. Давайте рассмотрим это на примере.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | import java.util.Comparator;    import org.junit.Test;  import static org.junit.Assert.assertEquals;    import org.junit.runner.JUnitCore;    import org.junit.runner.Description;  import org.junit.runner.Request;  import org.junit.runner.notification.RunListener;  import org.junit.runner.notification.Failure;    public class TestCalc {      // Обратите внимание на название - оно теперь сделано таким для соблюдения порядка        @Test      public void get001SumTest() {          Calc c = new Calc();          assertEquals(50, c.getSum(20, 30));      }        // Обратите внимание на название - оно теперь сделано таким для соблюдения порядка      @Test      public void get002SubtractionTest() {          Calc c = new Calc();          assertEquals(-10, c.getSubtraction(20, 30));      }        // Метод возвращает компаратор, который позволяет отсортировать методы в алфавитном порядке      private static Comparator forward() {          return new Comparator() {                public int compare(Object o1, Object o2) {                  Description d1 = (Description) o1;                  Description d2 = (Description) o2;                  return d1.getDisplayName().compareTo(d2.getDisplayName());              }          };      }        public static void main(String[] args) {          JUnitCore core = new JUnitCore();          core.addListener(new CalcListener());          // Обратите внимание на этот вызов - ниже немного комментариев          core.run(Request.aClass(TestCalc.class).sortWith(forward()));      }  }    // Ну здесь по идее должно быть все понятно  class CalcListener extends RunListener {        @Override      public void testStarted(Description desc) {          System.out.println("Started:" + desc.getDisplayName());      }        @Override      public void testFinished(Description desc) {          System.out.println("Finished:" + desc.getDisplayName());      }        @Override      public void testFailure(Failure fail) {          System.out.println("Failed:" + fail.getDescription().getDisplayName() + " [" + fail.getMessage() + "]");      }  } |

Итак, комментарии к вызову core.run(…). Здесь использован объект Request, который позволяет запускать не один, а сразу много классов для тестирования. В документации внимательно посмотрите на его методы и Вы сможете разобраться. Либо присылайте вопросы.

А тестовые данные как использовать ?

Вы можете определить набор тестовых данных, которые будут подаваться на вход тестирующего класса. Чтобы долго не растекаться по древу приведем пример. Комментарии смотрите после кода.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | import java.util.Arrays;  import java.util.Collection;    import org.junit.Test;  import static org.junit.Assert.assertEquals;  import org.junit.runner.JUnitCore;  import org.junit.runner.Description;  import org.junit.runner.RunWith;  import org.junit.runner.notification.RunListener;  import org.junit.runner.notification.Failure;  import org.junit.runners.Parameterized;  import static org.junit.runners.Parameterized.Parameters;    @RunWith(Parameterized.class)  public class TestCalc {      // Обратите внимание на данные в скобках - первый два - числа, которые складываем/вычитаем      // Вторые - это их сумма и разность        @Parameters      public static Collection data() {          return Arrays.asList(new Object[][]{                      {5, 3, 8, 2},                      {15, 10, 25, 5},                      {5, 10, 15, -5}                  });      }      int x1, x2, sum, sub;        public TestCalc(int x1, int x2, int sum, int sub) {          this.x1 = x1;          this.x2 = x2;          this.sum = sum;          this.sub = sub;      }        @Test      public void getSumTest() {          Calc c = new Calc();          assertEquals(sum, c.getSum(x1, x2));      }        @Test      public void getSubtractionTest() {          Calc c = new Calc();          assertEquals(sub, c.getSubtraction(x1, x2));      }        public static void main(String[] args) {          JUnitCore core = new JUnitCore();          core.addListener(new CalcListener());          core.run(TestCalc.class);      }  }    class CalcListener extends RunListener {        @Override      public void testStarted(Description desc) {          System.out.println("Started:" + desc.getDisplayName());      }        @Override      public void testFinished(Description desc) {          System.out.println("Finished:" + desc.getDisplayName());      }        @Override      public void testFailure(Failure fail) {          System.out.println("Failed:" + fail.getDescription().getDisplayName() + " [" + fail.getMessage() + "]");      }  } |

@RunWith — аннотация говорит о том, что мы будем запускать наш тест с помощью «запускателя» Parametrized.  
@Parameters — эта аннотация объявляет метод, который будет вызываться для получения данных.  
Теперь остановимся на следующем моменте — мы объявили конструктор для нашего теста, который принимает 4 параметра — да, да, Вы совершенно правы — по числу элементов в массиве. В конструкторе мы их запоминаем и в дальнейшем используем — смотрите код двух наших методов с @Test. Вот результаты нашего запуска:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Started:getSumTest[0](TestCalc)  Finished:getSumTest[0](TestCalc)  Started:getSubtractionTest[0](TestCalc)  Failed:getSubtractionTest[0](TestCalc) [expected:<2> but was:<1>]  Finished:getSubtractionTest[0](TestCalc)    Started:getSumTest[1](TestCalc)  Finished:getSumTest[1](TestCalc)  Started:getSubtractionTest[1](TestCalc)  Failed:getSubtractionTest[1](TestCalc) [expected:<5> but was:<4>]  Finished:getSubtractionTest[1](TestCalc)    Started:getSumTest[2](TestCalc)  Finished:getSumTest[2](TestCalc)  Started:getSubtractionTest[2](TestCalc)  Failed:getSubtractionTest[2](TestCalc) [expected:<-5> but was:<-6>]  Finished:getSubtractionTest[2](TestCalc) |

Как видите у нас тест выполнялся три раза. Я специально разделил вывод на три части – на самом деле получилось все вместе, но для понятности можно.  
Итак, возможно, что мне не удалось охватить все возможности JUnit, но даже этого вполне достаточно, чтобы быстро и удобно создавать и запускать тесты своей программы. Самое главное — ПРИДУМАТЬ ХОРОШИЙ ТЕСТ JUnit за Вас не сможет. Он может предоставить только удобные инстурменты для запуска и работы с тестом, но как и что проверять, увы, такой программы еще не написали. И вряд ли напишут.