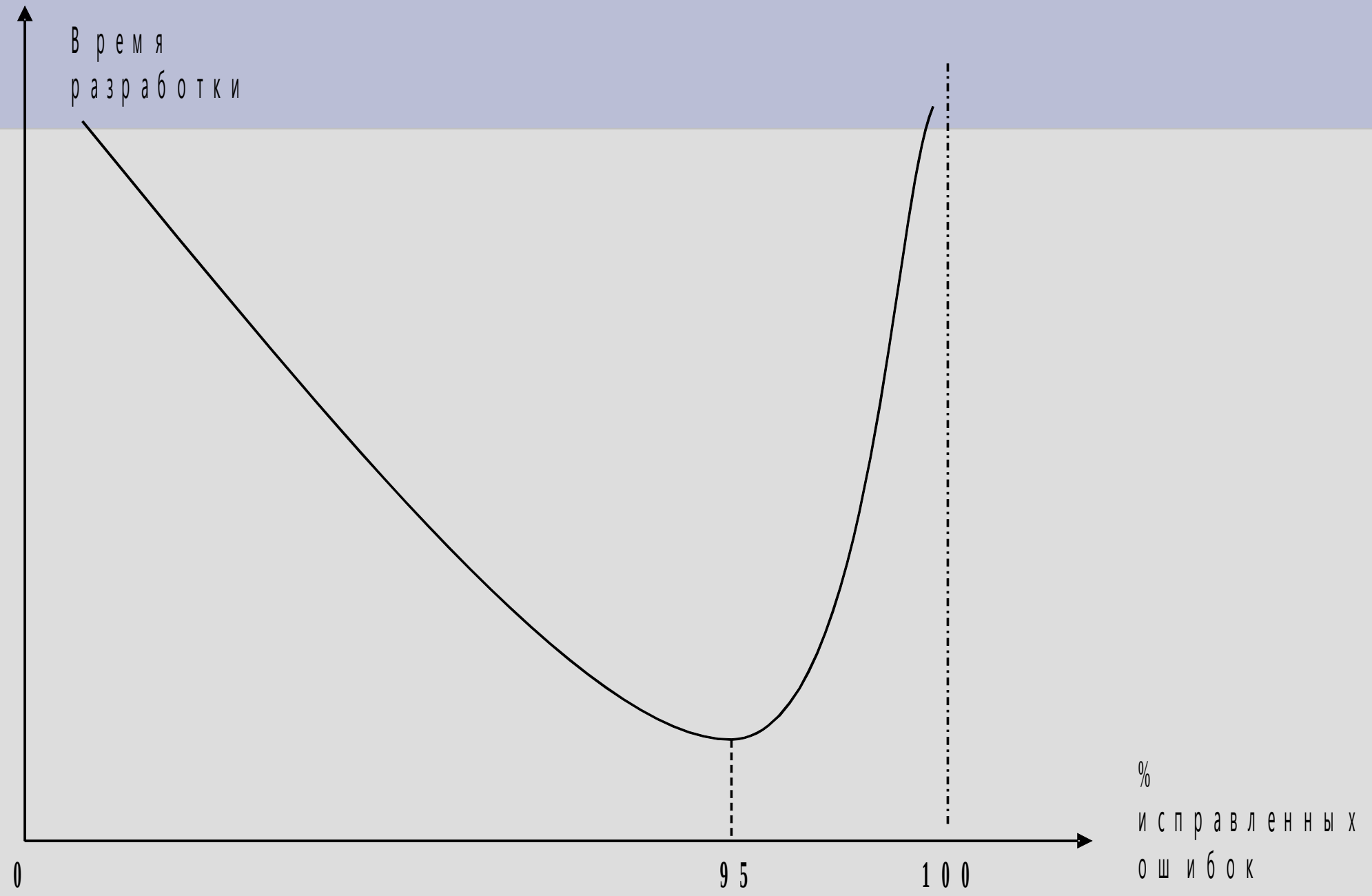
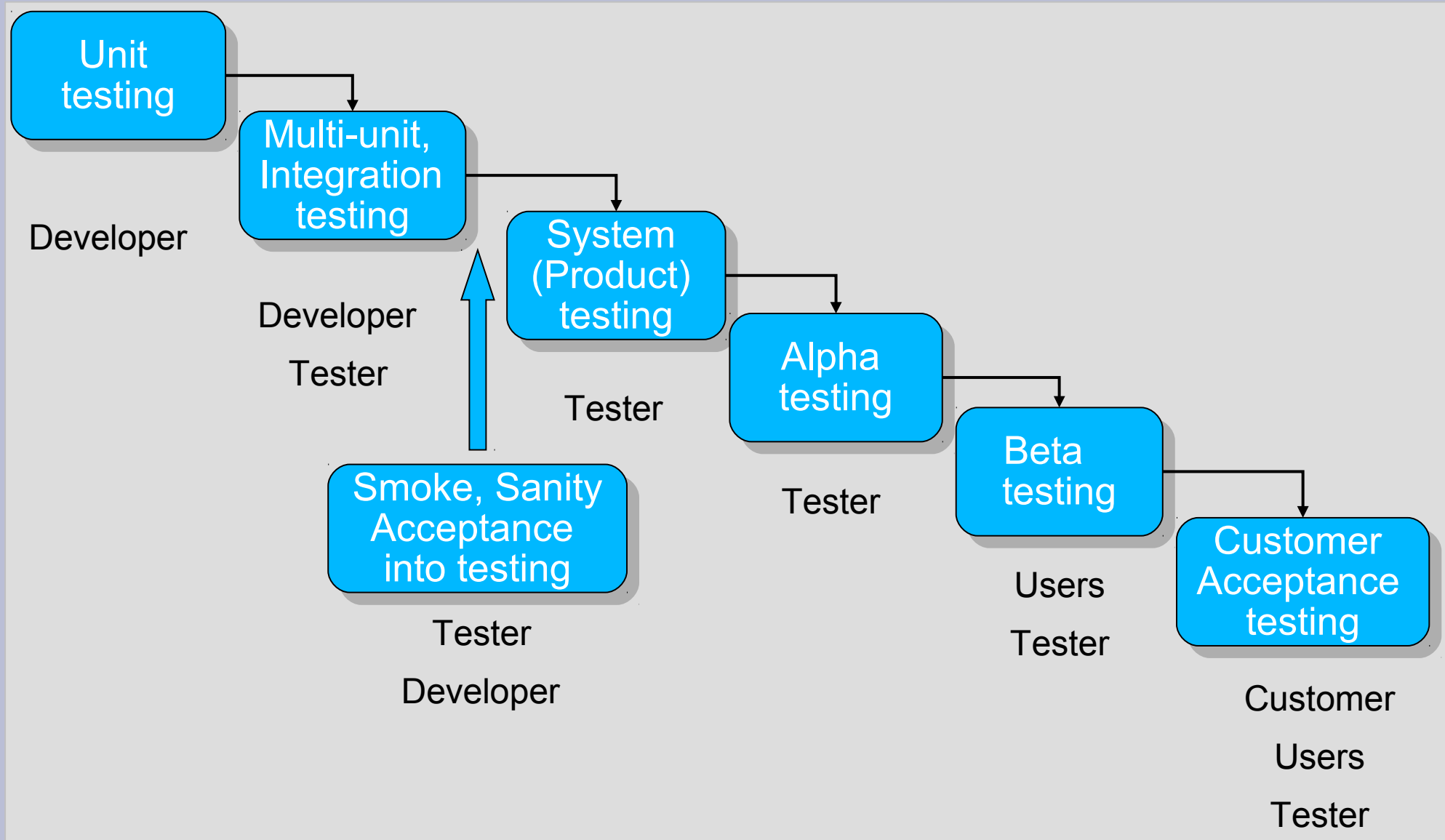
Модульное тестирования

Зачем нужно тестирование



"Applied Software Measurement", Capers Jones, 1991

Стадии (этапы) тестирования (Test stages)



Модульное тестирование
Unit testing

- Предназначено для проверки правильности отдельных модулей (функций, методов, библиотек), вне зависимости от их окружения не рассматривая взаимодействие с другими модулями.
- Необходимо чтобы данный модуль рассматривался изолированно используя вместо обращения к другим модулям заглушки (stubs), симуляторы (mocks).
- При этом проверяется, что, если модуль получает на данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны.
- Покрытие коды 85%
- Обычно используются Unit (JUnit, CUnit) тесты

Интеграционное тестирование
Multi-unit (integration) testing

Предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций.

Подходы в интеграционном тестировании:

- «большой взрыв» (Big Bang)
- восходящее (Bottom-up testing)
- нисходящее (Top-down testing)

Основное назначение интеграционного тестирования – протестировать взаимодействие модулей, то есть ***интерфейс*** между модулями.

Тестирование взаимодействия (интерфейса) Interface testing

Типы взаимодействия (интерфейса):

- Parameter interfaces – data passed from one procedure to another
- Shared memory interfaces (by reference) – block of memory is shared between procedures
- Procedural interfaces – sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces – sub-systems request services from other sub-systems

Ошибки взаимодействия:

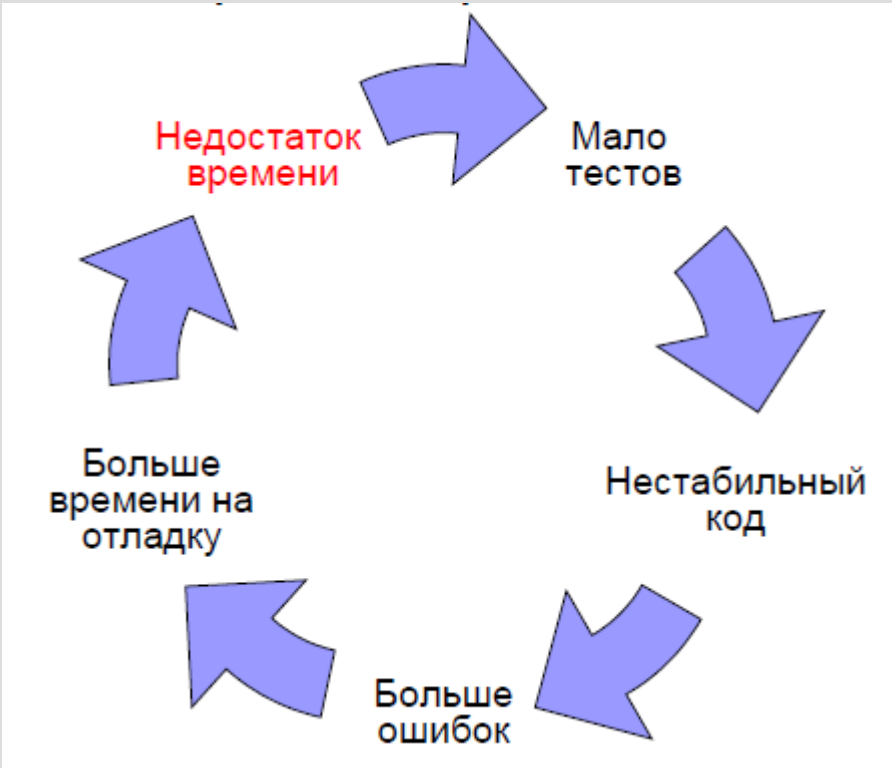
- Interface misuse – a calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- Interface misunderstanding – a calling component embeds assumptions about the behaviour of the called component which are incorrect
- Timing errors – the called and the calling component operate at different speeds and out-of-date information is accessed

Модульное тестирование

- Тестирование отдельных функций системы
- Как правило выполняется разработчиком модуля
- Может быть легко автоматизировано
- Закладывает основу для регрессионного тестирования приложения
- Является «документацией» к программе

«У меня нет времени на тесты»

- Написание тестов стабилизирует код и позволяет существенно сократить время отладки.



«Мой код и так отлично работает»

- Представим себе идеального разработчика. Он:
  - Прочитал все нужные книги
  - Не попал во все возможные ловушки
  - Знает все паттерны проектирования
  - Помнит наизусть детали реализации каждого из 363 классов своей системы
- Как Вы думаете, изменив 35-ю строчку 276 -го класса, сможет ли он точно предсказать, как это скажется на остальных 362?
- Не легче ли использовать набор надежных автоматических тестов в подобном случае – и не держать в голове лишние подробности?

Пример программы

```
public class CustomMath {

     public static int sum(int x, int y) {
        return x + y;
     }


    public static int division(int x, int y) {
        if (y == 0) {       //если делитель равен нулю
            throw new IllegalArgumentException("divider is 0 ");
        }                   //бросается исключение
        return x / y;
    }
}
```

Вариант модульного тестирования

```
public class CustomMath {
    public static void main(String[] args) {
        if (sum(1, 3) == 4) { //проверяем, что при сложении 1 и 3
                              //нам возвращается 4
        System.out.println("Test1 passed.");
        } else
            System.out.println("Test1 failed.");

        try {
        int z = division(1, 0);
        System.out.println("Test3 failed.");
        } catch (IllegalArgumentException e){
        //тест считается успешным, если при попытке деления на 0
        //генерируется ожидаемое исключение
        System.out.println("Test3 passed.");
    }
...
}
```

Что плохо

- Тесты неудобно хранить в самой программе
- Усложняет чтение кода
- Такие тесты сложно запускать
- Потенциально привносят свои ошибки в программу
- Тесты не относятся к бизнес-логике приложения и должны быть исключены из конечного продукта

JUnit

- Внешняя библиотека, подключенная к проекту может существенно облегчить разработку и поддержание модульных тестов
- Для каждого языка программирования реализация отличается
- JUnit – это библиотека, позволяющая проводить модульное тестирование Java приложений

Пример теста

- Автоматически сгенерированный тест

```
//подключение библиотек
import org.junit.Test;
import static org.junit.Assert.*;

public class CustomMathTest {
    //тестирующие методы должны иметь аннотацию “@Test”
    @Test
    public void testSum() {
        System.out.println("sum");
        int x = 0; int y = 0; int expResult = 0;
        int result = CustomMath.sum(x, y);
        //проверка условия на совпадение
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }
}
```

Asserts

- **assertTrue**(String *message*, Boolean *test*)
- **assertFalse**(String *message*, Boolean *test*)
- **assertNull**(String *message*, Object *object*)
- **assertNotNull**(String *message*, Object *object*)
- **assertEquals**(String *message*, Object *expected*, Object *actual*) (uses equals method)
- **assertSame**(String *message*, Object *expected*, Object *actual*) (uses == operator)
- **assertNotSame**(String *message*, Object *expected*, Object *actual*)

Обработка исключений

- Запуск теста для метода division(x, y) выдал ошибку:

```
Testcase: testDivision(javaapplication8.CustomMathTest):
Caused an ERROR
divider is 0
java.lang.IllegalArgumentException: divider is 0
        at javaapplication8.CustomMath.division(CustomMath.java:18)
        at javaapplication8.CustomMathTest.testDivision(CustomMathTest.java:35)
```

Обработка исключений

- Тест некорректно обрабатывает исключение
- Вместо сравнения числовых результатов надо ждать исключения

```
public void testDivision() {
    System.out.println("division");
    int x = 0; int y = 0; int expResult = 0;
    int result = CustomMath.division(x, y);
    assertEquals(expResult, result);
}
```

Обработка исключений (JUnit 3.0)

- Если наш метод работает правильно, то мы должны ожидать исключения и проверять его генерацию в тесте
- Если исключение не бросается значит наш метод работает неправильно

```
public void testDivision() {
    int x = 0; int y = 0; int expResult = 0;
    try {
            expResult = CustomMath.division(x, y);
            fail("Exception expected");
    } catch (IllegalArgumentException e){
        System.out.println("pass");
    }
}
```

Обработка исключений (JUnit 4.0)

- При необходимости тестирования конкретного исключения следует использовать предлагаемую в версии JUnit 4 аннотацию @Test с параметром expected. Этот параметр предназначен для представления типа исключения, которое данный тест должен выдавать в процессе исполнения.
- Тест завершится успешно лишь в том случае, если возникнет исключительная ситуация.

```
…
@Test(expected=NotSuchMarkException.class)
        public void stepUpCoefficientForElevenTest() throws NotSuchMarkException{
                double expected=1;
                double actual=scholarshipCalculator.stepUpCoefficientCalculate(11);
                assertEquals("Coefficient for mark 11 ",expected, actual);
                }
```

JUnit – Fixtures

Фикстура (Fixture) - состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Это может быть набор каких-либо объектов, состояние базы данных, наличие определенных файлов и т.д.

- **@BeforeClass** – запускается перед любым тестовым методом класса один раз.
- **@Before** – запускается перед каждым тестовым методом.
- **@After** – запускается после каждого метода.
- **@AfterClass** – запускается после того, как отработали все тестовые

JUnit – Fixtures

```
public class ScholarshipCalculatorTest {
		private ScholarshipCalculatorImpl scholarshipCalculator;
		@Before
		public void initScholarshipCalculator(){
				scholarshipCalculator=new ScholarshipCalculatorImpl();
		}
		@After
		public void clearScholarshipCalculator(){
				scholarshipCalculator=null;
		}
		@Test
		public void stepUpCoefficientForFiveTest(){
		double expected=1.5;
		double actual=scholarshipCalculator.stepUpCoefficientCalculate(5);
				assertEquals("Coefficient for mark 5 ",expected, actual);
		}
		@Test
		public void stepUpCoefficientForTwoTest(){
		double expected=1;
		double actual=scholarshipCalculator.stepUpCoefficientCalculate(3);
				assertEquals("Coefficient for mark 3 ",expected, actual);
		}
}
```

JUnit - timeout

- В версии JUnit 4 в качестве параметра тестового сценария может быть использовано значение лимита времени (timeout). Значение timeout представляет максимальное количество времени, отводимого на исполнение данного теста: при превышении этого лимита времени тест завершается неудачей.
- Организовать тестирование с ограничением по времени несложно – для создания автоматизированного теста с ограничением по времени достаточно после аннотации @Test указать значение параметра timeout.

JUnit - timeout

```
public class ScholarshipCalculatorTest {

        @Test(timeout = 10)
        public void scholarshipCalculateTest() {
                IScholarshipCalculator scholarshipCalculator = new
ScholarshipCalculatorImpl();
double basicScholarship = ScholarshipCalculatorImpl.basicScholarship;
                for (int i = 1; i < 5000; i++) {
                        double stepUpCoefficient = 1 / i;
                        double expected = basicScholarship * stepUpCoefficient;
                        double actual = scholarshipCalculator
                                        .scholarshipCalculate(stepUpCoefficient);
                        org.junit.Assert.assertEquals(expected, actual);
                }
        }
    }
```

JUnit - Ignore

- В версии JUnit 4 предусмотрена аннотация @Ignore, которая заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Можно также вставить комментарий к вашему решению об игнорировании теста – для разработчиков, которые могут впоследствии случайно столкнуться с этим тестом.

Разработка через тестирование

- **Разработка через тестирование** - процесс разработки программного обеспечения, который предусматривает написание и автоматизацию модульных тестов еще до момента написания соответствующих классов или модулей. Это гарантирует, что все обязанности любого элемента программного обеспечения определяются еще до того, как они будут закодированы.

Подход «Сначала тесты, затем код» Test Driven Development (TDD)

- Каждой строчке кода предшествует неработающий тест (неработающий — не значит неправильный !!!)
- После написания кода тест должен пройти
- Как только тест проходит, то он усложняется, либо добавляется еще один тест
- Затем вновь совершенствуется код
- Разработка проходит маленькими итерациями, с чередованием написания кода и его тестирования
- При интеграции кода с другими частями системы все тесты должны проходить успешно

Преимущества TDD

- Каждая функция протестирована
  - Если изменение «ломает» старую функциональность, падение тестов служит индикатором проблемы
- Тесты документируют код, поскольку показывают, какие результаты и исключения следует ожидать от каждого модуля
- Итерации разработки очень коротки (порядка 10 минут)
  - Несложно отследить какие изменения это повлекли
- Дизайн приложения заметно упрощается
  - Сложные методы сложно тестировать (и поддерживать)

Рекомендации по написанию тестов

- Название тестового метода
- Размер теста
- Ожидаемый результат
- Тестовые данные
- Исключения

Название тестового метода

- Имя теста должно отражать:
  - Тестируемую функциональность
  - Возможно, условия тестирования

Непонятно	Понятно
test1	AddUser
testAddUserThrowException	AddUserWithoutPasswordThrowException

Название тестового метода Given/When/Then

- given - the precondition or setup for a test or set of tests
- when - events whose effect you're testing/specifying
- then - assertions/specifications of expected behaviour

```
@Test
public void shouldDoSomethingCool() throws Exception {
    //given

    //when

    //then

}
```

Название тестового метода

[MethodName_StateUnderTest_ExpectedBehavior]

[MethodName_StateUnderTest_ExpectedBehavior]

Examples:

public void sum_NegativeNumberAs1stParam_ExceptionThrown()

public void sum_NegativeNumberAs2ndParam_ExceptionThrown ()

public void sum_simpleValues_Calculated ()

public void parse_OnEmptyString_ExceptionThrown()

public void parse_SingleToken_ReturnsEqualToeknValue ()

Размер теста

- Тестовый метод должен быть коротким
  - Иногда полезно выносить дополнительные проверки во вспомогательные методы, чтобы улучшить читаемость теста
- Количество проверок (assert) должно быть
- минимальным
  - Иначе по падению теста сложно будет найти причину ошибки
- Каждый тест должен покрывать одну единицу бизнес-логики. Это может быть:
  - Простой метод
  - Один из исходов конструкции if..else
  - Один из случаев (case) блока switch
  - Исключение, обрабатываемое блоком try…catch
  - Исключение, генерируемое (throw) в методе

Ожидаемый результат

- Ожидаемый результат должен быть **<u>константой</u>**
- Не следует в тесте повторять тестируемую логику, подсчитывая результат

```
public void testSum1() {
        System.out.println("sum");
        int x = 5;
        int y = 6;
        int expResult = 11;     //Правильно
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
    }

public void testSum2() {
        System.out.println("sum");
        int x = 5;
        int y = 6;
        int expResult = x+y;   //Неправильно
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
    }
```

Тестовые данные

- Тестовые данные и ожидаемый результат должны быть рядом
- Если в приведенном примере объект user удобнее создавать вне тестового метода (например, чтобы избежать дублирования кода), следует использовать именованные константы

Тестовые данные

```
public void testIsPasswordValid()    {
      assertTrue(user.isPasswordValid("abcdef"));
      //понять,правильно ли написан тест, можно лишь отыскав где
      //создается объект user
      assertFalse(user.isPasswordValid("123456"));
}

public void testIsPasswordValid() {
      User user = new User("Name", "abcdef");
      assertTrue(user.isPasswordValid("abcdef"));
      //здесь все понятно
      assertFalse(user.isPasswordValid("123456"));
}

public void testIsPasswordValid() {
      assertTrue(user.isPasswordValid(CORRECT_PASSWORD));
      //альтернативный вариант с использованием именованных
      //констант
      assertFalse(user.isPasswordValid(WRONG_PASSWORD));
}
```

Исключения

- Если получение исключения не обозначает удачное завершение теста, рекомендуется пробрасывать его их тестового метода, а не обрабатывать блоком try…catch.
- Само исключение более информативно, чем сообщение
- Не рекомендуется использовать общий класс (Exception), так как это затрудняет чтение и поддержку теста

Пример

```
public void testRetrieveUser() {
      try{
            assertNotNull(manager.retrieveUser("name","password"));
      } catch (WrongPasswordException e) {
            fail("Exception occurred: " + e.getMessage());
            e.printStackTrace();
      }
}

public void testRetrieveUser() throws Exception {
      assertNotNull(manager.retrieveUser("name", "password"));
}

public void testRetrieveUser() throws WrongPasswordException {
      assertNotNull(manager.retrieveUser("name","password"));
} //оптимальный вариант
```

Итого

- **Поощрение изменений**
  - Юнит-тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.
- **Упрощение интеграции**
  - Юнит-тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируются отдельные части программы, затем программа в целом.
- **Документирование кода**
  - Юнит-тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

Итого

- **Отделение интерфейса от реализации**
  - Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных. В ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.
    - **Моки, Мок-объекты (MockObjects)** - автоматически генерируемые заглушки, которые могу выступат в роли реальных объектов. Поведением моков можно управлять непосредственно в тесте.
- **Баг-трэкинг**
  - В случае обнаружения бага для него можно (даже рекомендуется) создать тест для выявления повторения подобной ошибочной ситуации при последующем изменении кода.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated

A good unit test is

http://artofunittesting.com/unit-testing-review-guidelines/

- Able to be fully automated
- Has full control over all the pieces running (Use mocks or stubs to achieve this isolation when needed)
- Can be run in any order if part of many other tests
- Runs in memory (no DB or File access, for example)
- Consistently returns the same result (You always run the same test, so no random numbers, for example. save those for integration or range tests)
- Runs fast
- Tests a single logical concept in the system
- Readable
- Maintainable
- Trustworthy (when you see its result, you don’t need to debug the code just to be sure)

Как проверить что тесты хорошие?

Задания

- Простые задачки (сортировка массива за n*log(n), числа Фибоначчи, простые делители, …) на которые нужно будет написать unit-тесты
- Реализовать класс Complex и тесты к нему
- Именовать тесты согласно правилу [MethodName_StateUnderTest_ExpectedBehavior]

Обязательная «литература»

- Understanding TDD
  - http://osherove.com/videos/2009/8/25/understanding-test-driven-development.html
- Unit Testing Best Practices
  - http://osherove.com/videos/2009/8/25/unit-testing-best-practices.html
- http://artofunittesting.com

Обязательная «литература»

- Understanding TDD
  - http://osherove.com/videos/2009/8/25/understanding-test-driven-development.html
- Unit Testing Best Practices
  - http://osherove.com/videos/2009/8/25/unit-testing-best-practices.html
- http://artofunittesting.com

Литература

- http://refcardz.dzone.com/refcardz/junit-and-easymock
- http://junit.sourceforge.net/doc/testinfected/testing.htm
- http://junit.sourceforge.net/doc/faq/faq.htm
- http://junit.sourceforge.net/javadoc/index.html
- http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html
- http://www.ibm.com/developerworks/ru/edu/j-junit4/index.html
- http://ru.wikipedia.org/wiki/Юнит-тестирование
- http://litvinyuk.com/articles/junit.htm
- http://wiki.agiledev.ru/doku.php?id=tdd:glossary