Лабораторная работа №3

Компонентное или модульное тестирование (юнит-тестирование)

*Цель работы:* научиться технике модульного тестирования с помощью фреймворка JUnit.

*Отчет по лабораторной работе:* тестируемая программа на языке Java, набор модульных тестов к ней.

Теоретическая часть

# Определения

**Модульное тестирование**, или **юнит-тестирование** (англ. unit testing) - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к **регрессии**, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

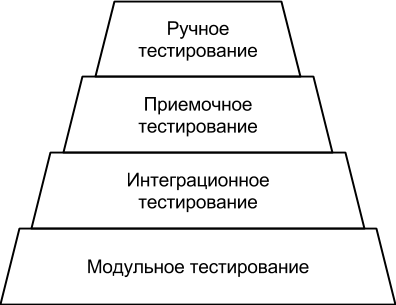
**Mock-объект** (от англ. mock object, буквально: объект-пародия, объект-имитация) - в ООП тип объектов, реализующих заданные аспекты моделируемого программного окружения.

Mock-объект представляет собой конкретную фиктивную реализацию интерфейса, предназначенную исключительно для тестирования. В процедурном программировании аналогичная конструкция называется «dummy» (англ. — заглушка). Это может быть функция, выдающая константу или случайную величину из допустимого диапазона значений.

# Назначение модульного тестирования

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Этот тип тестирования обычно выполняется программистами.

Качественно выполненное модульное 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.

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх» (см. рисунок): сначала тестируются отдельные части программы, а затем программа в целом.

Для Java отдельным компонентом, из которого состоит программа, является **класс**. Если каждый программист проверит, что поведение его класса (модуля) соответствует задуманному, то и программа, состоящая из таких оттестированных классов, скорее всего, будет работать как задумано.

Что значит — «программист проверит»? Это не означает, что он сделает это вручную. Вручную это делать неэффективно. Проверит — означает, что программист напишет небольшую программу для тестирования поведения своего юнита (класса).

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Классы-клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс активно 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 “класс-пустышку”, или mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

Таким образом, юнит-тестирование – это первый “бастион” на борьбе с ошибками в ПО. За ним следуют еще интеграционное, приемочное и, наконец, ручное тестирование, в том числе «свободный поиск».

# Основные правила составления юнит-тестов

Для того, чтобы тесты было легче разрабатывать и поддерживать, нужно придерживаться следующих основных правил. Юнит-тесты должны:

* Быть достоверными;
* Не зависеть от окружения, на котором они выполняются. Особенно это важно для кроссплатформенных программ, в числе которых веб-приложения;
* Легко поддерживаться;
* Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется);
* Соблюдать соглашения об именовании;
* Запускаться регулярно в автоматическом режиме.

## Логическое расположение тестов в системе контроля версий

Тесты должны быть частью контроля версий. Они могут быть организованы по-разному. Общая рекомендация: если приложение монолитное, разместить все тесты в папке Tests; если в приложении много разных компонентов, можно хранить тесты в папке каждого компонента.

## Способ именования проектов и классов с тестами

Одна из лучших практик: добавить к каждому проекту его собственный тестовый проект. Если есть части системы <PROJECT\_NAME>.Core, <PROJECT\_NAME>.Bl и <PROJECT\_NAME>.Web, то рационально добавить еще <PROJECT\_NAME>.Core.Tests, <PROJECT\_NAME>.Bl.Tests и <PROJECT\_NAME>.Web.Tests. Рекомендуется использовать такой же способ именования для тестовых классов. Если есть класс ProblemResolver, то при написании юнит-теста для этого класса необходимо дообавить в тестовый проект (или папку для тестов) класс ProblemResolverTests. **Каждый тестирующий класс должен тестировать только одну сущность.**

# Составление Unit-тестов с использованием фреймворка JUnit 4

В данной лабораторной работе необходимо создать Java-проект, который будет подвергнут модульному тестированию. Для обеспечения модульного тестирования будет использоваться библиотека JUnit 4 (<http://junit.org/>).

JUnit представляет из себя библиотеку классов. Для демонстрации основных возможностей этой библиотеки, можно написать примитивный класс:

public class Salary {

private int value;

private String type;

// конструктор объекта класса “Зарплата”, использующий числовое значение

// и строку для её типа

public Salary(int value, String type){

this.value = value;

this.type = type;

}

// метод возвращает новый объект класса “Зарплата”, в котором значение

// зарплаты увеличилось на величину зарплаты аргумента

public Salary add(Salary s){

return new Salary(value + s.getValue(), type);

}

public int getValue(){

return value;

}

}

Объявление теста сводится к маркировке тестового метода аннотацией @Test. Создадим несколько тестовых методов:

import static org.junit.Assert.\*;

import org.junit.Test;

public class TestSalary {

@Test

public void testAdd() {

Salary m1 = new Salary(12, "USD");

Salary m2 = new Salary(14, "USD");

Salary expected = new Salary(26, "USD");

Salary result = m1.add(m2);

assertFalse(expected.equals(result));

}

}

Метод *assertFalse* проверяет, является ли результат выражения в скобках неверным. При запуске вышеприведенного теста он пройдет успешно, т.к. результатом вычисления выражения *expected.equals(result)* является *false*.

В классе *org.junit.Assert* предусмотрены и другие методы:

* *assertEquals(int1, int2)* или утверждение эквивалентности. Проверяет на равенство два значения любого примитивного типа;
* *assertFalse(condition)*, *assertTrue(condition)* или булевые утверждения. Вместо “condition” необходимо вставить проверяемое условие;
* *assertNull(obj)*, *assertNotNull(obj)* относятся к null-утверждениям и проверяют содержимое объектной переменной на null значение;
* *assertSame(obj1, obj2)* утверждение позволяет сравнивать объектные переменные.

Для каждого из assert-ов возможно добавить первым параметром строку, которая выведется, если тест провалится: *assertEquals(“Test is failed”, int1, int2)*.

Для аннотации @Test существуют дополнительные опции. Например:

@Test(expected = Exception.class)

public void testDiv(){

Salary m1 = new Salary(12, "USD");

Salary m2 = new Salary(0, "USD");

int result = m1.getValue() / m2.getValue();

}

“expected = Exception.class” означает, что ожидается появление исключения *Exception*. Если исключение не будет выброшено, то такое поведение тестируемой функции будет неверным и тест провалится.

Если тест по какой-либо серьезной причине нужно отключить (например, этот тест постоянно валится, а исправлять его пока некогда) его можно зааннотировать с помощью *@Ignore*:

import org.junit.Ignore;

import org.junit.Test;

@Ignore

@Test(timeout = 1000)

public void testAdd() {

//код

}

Также, если поместить эту аннотацию перед объявлением класса, то все тесты в этом классе будут отключены. Помимо этого есть также интересная аннотация *@Test(timeout = 1000)*. По истечении указанного в скобках времени, если тест не пройден, он считается неудачным. Время указывается в миллисекундах.

В JUnit для задания определенных стартовых условий могут пригодиться так называемые **фикстуры**. Под этим термином следует понимать состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Например, это может быть набор каких-либо объектов или состояние базы данных. Фикстуры помогают многократно использовать программный код за счет правила, которое гарантирует исполнение определенной логики до или после исполнения теста. Фикстуры указываются через аннотации: *@Before*, *@After*, *@BeforeClass*, *@AfterClass*.

*@Before* используется для выполнения множества предварительных условий перед выполнением теста. Например, если есть необходимость записать данные в БД или создать пользователя перед выполнением теста. Метод, помеченный *@Before* будет выполняться перед выполнением *каждого* теста в классе.

Метод, помеченный *@After*, запускается после выполнения *каждого* теста. Например, если нужно очищать переменные после выполнения каждого теста, то этой аннотацией можно маркировать метод, имеющий необходимый код. Более того, можно маркировать одновременно несколько методов аннотациями *@Before* и *@After*. Однако, следует иметь в виду, что эти методы могут запускаться в различном порядке. Для задания многократных фикстур используются аннотации *@Before* и *@After*:

import org.junit.After;

import org.junit.Before;

@Before

public void setup(){

Money m1 = new Money(12, "USD");

Money m2 = new Money(14, "USD");

}

@After

public void setup(){

m1 = null;

m2 = null;

}

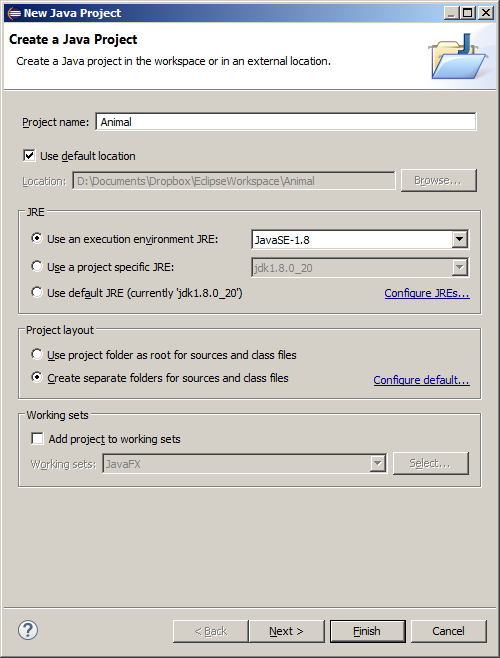
Также есть такие аннотации, как *@BeforeClass*, *@AfterClass* (так называемые однократные фикстуры). Они необходимы, если необходимо вызвать фикстуру всего один раз.

Практическая часть

# Пример создания проекта с обеспеченным модульным тестированием

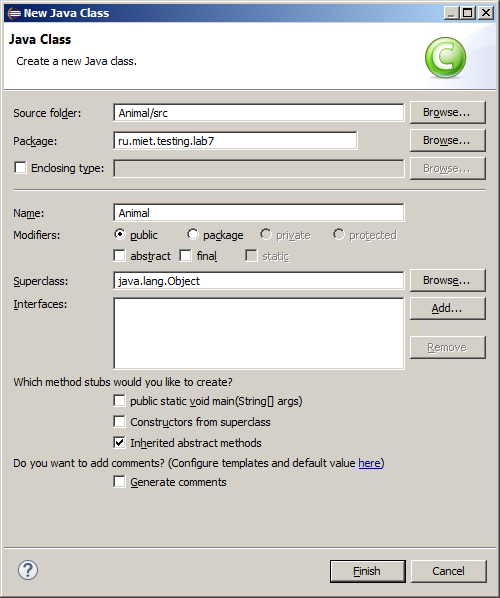
Далее описано создание Java-проекта, реализующего класс Animal. Класс Animal обладает свойствами name, height и weight. Этот класс также реализует возможность порождать новый экземпляр класса Animal, используя два ранее созданных экземпляра класса Animal.

В Eclipse IDE перейти в меню: File > New > Java Project. Ввести имя проекта “Animal”.



Нажать “Finish”. В Eclipse IDE будет добавлен новый проект с названием “Animal”.

В контекстном меню проекта выбрать New > Class. Ввести имя пакета в поле “Package”: *ru.miet.testing.lab7*; ввести имя класса в поле “Class”: *Animal*.



Нажать “Finish”. В Eclipse IDE должна открыться созданная заготовка для класса Animal. Ниже приведена готовая реализация класса Animal. Необходимые пояснения даны в комментариях.

package ru.miet.testing.lab7;

public class Animal {

// имя животного

private String name;

// рост животного

private double height;

// вес животного

private double weight;

// получение имени животного

public String getName() {

return name;

}

// задание имени животного

public void setName(String name) {

this.name = name;

}

// получение высоты животного

public double getHeight() {

return height;

}

// задание высоты животного

public void setHeight(double height) {

this.height = height;

}

// получение веса животного

public double getWeight() {

return weight;

}

// задание веса животного

public void setWeight(double weight) {

this.weight = weight;

}

// конструктор, создающий новый объект типа "Животное", используя имя, рост

// и вес

public Animal(String name, double height, double weight) {

if (name == null || height <= 0 || weight <= 0) {

// если в конструктор были переданы некорректные данные, необходимо

// "выкинуть" исключение подходящего типа

throw new IllegalArgumentException();

}

this.name = name;

this.height = height;

this.weight = weight;

}

// статический метод, создающий новое животное: оно наследует свой рост от

// первого предка, а вес от второго предка

public static Animal createNewAnimal(Animal firstParent, Animal secondParent) {

String nameNew = firstParent.getName() + " " + secondParent.getName();

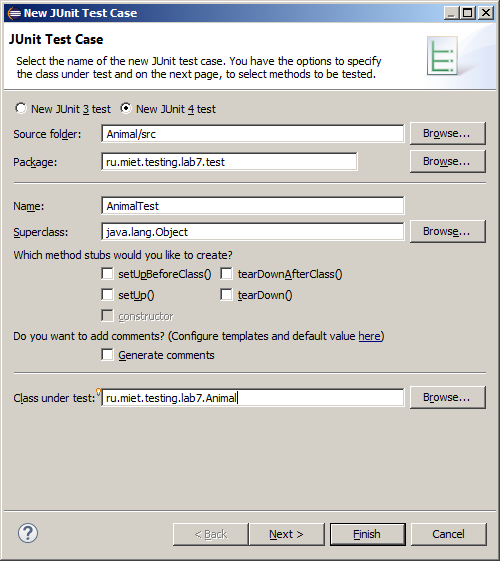
return new Animal(nameNew, firstParent.getHeight(),

secondParent.getWeight());

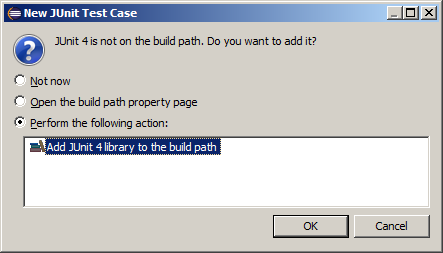
}

}

Далее необходимо написать модульный тест для класса Animal. Заготовку для написания теста можно получить следующим образом. В контекстном меню проекта выбрать New > JUnit Test Case и заполнить следующие поля. Package: *ru.miet.testing.lab7.test*. Name: *AnimalTest*. Class under test: *ru.miet.testing.lab7.Animal*.



Нажать “Finish”. Далее появится диалоговое окно с предложением добавить в проект библиотеку JUnit4. Необходимо нажать “ОК”.



Затем должна открыться заготовка для класса AnimalTest. Ниже приведен её исходный код с добавленными комментариями.

package ru.miet.testing.lab7.test;

import static org.junit.Assert.\*;

import org.junit.Test;

// класс, который реализует модульное тестирование класса Animal

public class AnimalTest {

/\*

\* Методы, помеченные аннотациями, могут называться как угодно (учитывая

\* ограничения языка Java), однако необходимо принимать во внимание

\* общепринятые соглашения разработчиков относительно именования методов

\*/

// аннотация @Test показывает, что данный метод представляет собой

// тест-кейс, что позволяет в дальнейшем его соответственно запускать

@Test

public void test() {

// тест-кейс сообщает об ошибке в процессе работы с сообщением

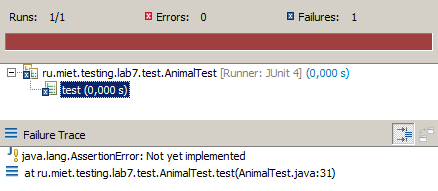
// "Not yet implemented"

fail("Not yet implemented");

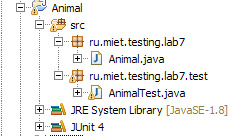
}

}

После запуска посредством контекстного меню: *Run As > JUnit Test* тест будет показывать, что он не пройден, что обуславливается непосредственным применением функции *fail*.



Структура получившегося проекта выглядит следующим образом:



Реализация тестирующего класса показана на примере. Тест организован следующим образом: в методе *testAnimalGetSetInitializing()* проверяется, корректно ли работают get и set методы класса *Animal*. В методе *testAnimalCreation()* проверяется, корректно ли создается экземпляр *Animal* из двух других. Остальные методы проверяют, корректно ли обрабатываются данные, если они являются неверными.

package ru.miet.testing.lab7.test;

import static org.junit.Assert.\*;

import org.junit.Test;

import ru.miet.testing.lab7.Animal;

public class AnimalTest {

private Animal animalMoo = new Animal("Moo", 3.86, 2.74);

private Animal animalQuack = new Animal("Quack", 0.4, 13.25);

private Animal animalMeow = new Animal("Meow", 15.744, 3.14);

@Test

public void testAnimalGetSetInitializing() {

// проверка корректного срабатывания конструктора

assertTrue(animalMoo.getName().equals("Moo"));

assertTrue(animalMoo.getHeight() == 3.86);

assertTrue(animalMoo.getWeight() == 2.74);

// проверка задания значений через set-методы (сеттеры)

animalMoo.setName("MooOld");

animalMoo.setHeight(3.56);

animalMoo.setWeight(3.12);

assertTrue(animalMoo.getName().equals("MooOld"));

assertTrue(animalMoo.getHeight() == 3.56);

assertTrue(animalMoo.getWeight() == 3.12);

}

@Test

public void testAnimalCreation() {

// проверка корректности создания нового животного из двух других

Animal animalNew = Animal.createNewAnimal(animalQuack, animalMeow);

assertTrue(animalNew.getName().equals("Quack Meow"));

assertTrue(animalNew.getHeight() == 0.4);

assertTrue(animalNew.getWeight() == 3.14);

}

// тесты, проверяющие, бросается ли исключение "Аргумент неверного типа" при

// создании нового животного c некорректными параметрами (меньше нуля и

// равными нулю)

@Test(expected = IllegalArgumentException.class)

public void testAnimalNullName() {

Animal animalNullName = new Animal(null, 3.2, 4.17);

}

@Test(expected = IllegalArgumentException.class)

public void testAnimalNegativeHeight() {

Animal animalNegativeHeight = new Animal("Animal", -0.17, 4.17);

}

@Test(expected = IllegalArgumentException.class)

public void testAnimalNegativeWeight() {

Animal animalNegativeWeight = new Animal("Animal", 0.75, -65);

}

@Test(expected = IllegalArgumentException.class)

public void testAnimalZeroHeight() {

Animal animalZeroHeight = new Animal("Animal", 0, 4.17);

}

@Test(expected = IllegalArgumentException.class)

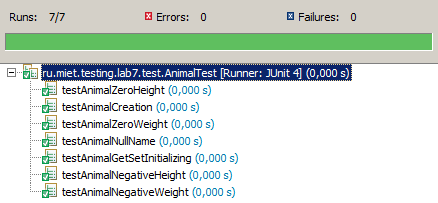
public void testAnimalZeroWeight() {

Animal animalZeroWeight = new Animal("Animal", 0.75, 0);

}

}

Запуск этого теста через Run As > JUnit Test приведет к успешному завершению теста.



# Задание к выполнению

## Общие требования

Реализовать программу (на языке Java), указанную в своём варианте. Написать модульные тесты c использованием JUnit 4 для каждого из классов программы. Программа должна быть написана на языке Java с соблюдением Java Conventions (<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>). Должно быть обеспечено максимальное покрытие тестами того класса или участка кода, модульное тестирование которого проводится. При составлении тест-кейсов рекомендуется использовать техники тест-дизайна (эквивалентное разделение, анализ граничных значений, причина / следствие, предугадывание ошибки). Обработка исключительных ситуаций должна быть реализована только через механизм Java Exceptions (<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>).

## Вариант №1, 6, 11, 16, 21, 26

Реализовать класс “Книга” со следующими полями: фамилия автора, имя автора, название, год выпуска книги, тираж. Обеспечить возможность установки и считывания всех полей. Добавить возможность добавлять в качестве соавтора(ов) автора другой книги. Хранение списка авторов книги можно обеспечить с помощью класса ArrayList.

## Вариант №2, 7, 12, 17, 22, 27

Реализовать класс “Смартфон” со следующими полями: производитель, название модели, диагональ дисплея, операционная система, толщина корпуса. Обеспечить возможность установки и считывания всех полей. Добавить возможность “связывания смартфонов”, которое возможно только в случае если у двух смартфонов один и тот же производитель, либо одинаковая операционная система.

## Вариант №3, 8, 13, 18, 23, 28

Реализовать класс “Строка” со следующими полями: длина строки, массив символов в этой строке, язык строки. Обеспечить возможность установки и считывания всех полей, кроме массива символов. Добавить возможность добавлять к данным объекта “Строка” данные из другого объекта.

## Вариант №4, 9, 14, 19, 24, 29

Реализовать класс “Железнодорожный вагон” со следующими полями: ссылка на подсоединенные вагоны (спереди и сзади), год производства, завод-изготовитель, страна-изготовитель, масса. Обеспечить возможность установки и считывания всех полей. Добавить возможность “соединения” вагонов, т.е установка ссылок на предыдущие или следующие вагоны. Добавить контроль неприсоединения как слишком старых вагонов, так и других вагонов к слишком старым вагонам (произведенные ранее 1950 г).

## Вариант №5, 10, 15, 20, 25, 30

Реализовать класс “Страна” со следующими полями: название, имя лидера, население, площадь, столица, язык. Обеспечить возможность установки и считывания всех полей. Добавить возможность “аннексии” страной с большей территорией страны с более маленькой территорией. Если более маленькая страна пытается аннексировать страну с большей территорией, то она сама аннексируется.

Использованные статьи

1. <https://ru.wikipedia.org/wiki/%CC%EE%E4%F3%EB%FC%ED%EE%E5_%F2%E5%F1%F2%E8%F0%EE%E2%E0%ED%E8%E5>
2. <https://ru.wikipedia.org/wiki/Mock-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82>
3. <http://habrahabr.ru/company/etnasoft/blog/169381/>
4. <http://dev64.wordpress.com/2013/04/18/unit-testing/>
5. <http://www.quizful.net/post/jUnit4_common_information>