**Лабораторная работа № 5**

**«Разработка эффективного кода. Функциональное тестирование приложений Java. »**

**1. Цель работы:**

Изучить методологии функционального тестирования приложений Java.

**2. Методические указания**

Пусть есть класс, реализующий несколько математических функций:

```
public class CustomMath {

    public static int sum(int x, int y) {
        return x + y; //возвращает результат сложения двух чисел
    }

    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; //возвращает результат деления
    }
}
```

**Замечание**

Иногда требуется снабжать программу модульными тестами.

Тесты неудобно хранить в самой программе:

1. Усложняет чтение кода.
2. Такие тесты сложно запускать.
3. Тесты не относятся к бизнес-логике приложения и должны быть исключены из конечного продукта.

Внешняя библиотека, подключенная к проекту, может существенно облегчить разработку и поддержание модульных тестов. Наиболее популярная библиотека для Java – JUnit.

**Вариант модульного тестирования без библиотеки**

Некоторые проверки можно поместить в сам класс. Доработаем класс CustomMath

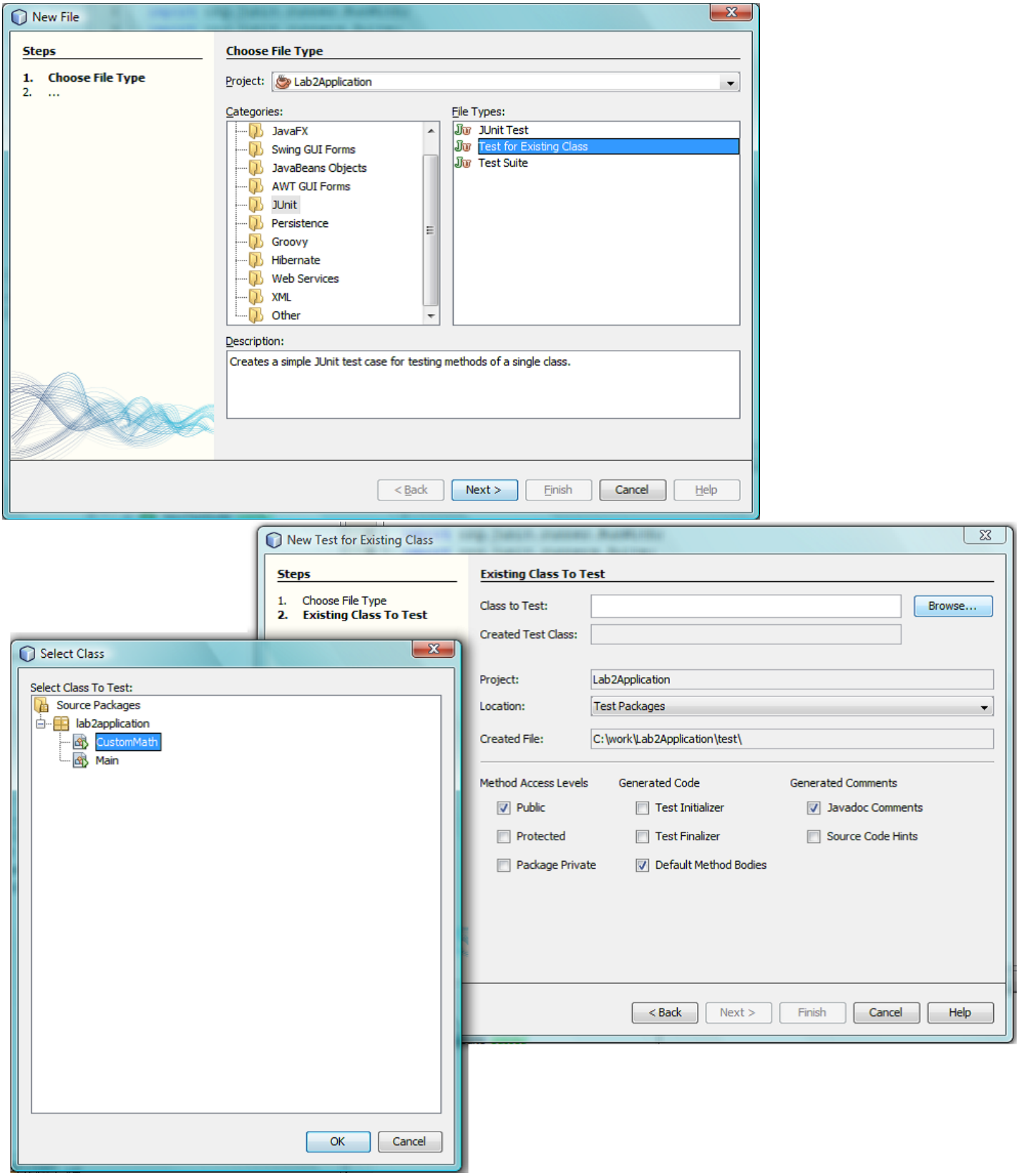
```
public class CustomMath {

    public static int sum(int x, int y) {...}

    public static int division(int x, int y) {...}

    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.");
        }
    }
}
```

**Установка JUnit**

JUnit может быть использован для любого Java-приложения. Сайт проекта www.junit.org. Библиотека входит в состав большинства интегрированных сред разработки, в том числе NetBeans.

**Создание тестового модуля**

Создание тестового модуля по шаблону может быть произведено с помощью мастера. Тесты JUnit будут располагаться в ветке Test Packages проекта. Структура папок в Test Packages в общем случае дублирует папки классов Source Packages. Выберите в меню File->New File->… в разделе JUnit пункт Тест для существующего класса («Test for Existing Class»).




В данном случае будут созданы тесты для класса CustomMath. Настройки оставим по умолчанию: доступ к методам Public, наполнение методов по умолчанию, комментарии Javadoc. Javadoc - форма организации комментариев в коде с использованием ключевых слов, по которым NetBeans определяет существенную информацию. Если класс оформлен с использованием Javadoc – по нему может быть автоматически создана документация, а также работать контекстная подсказка NetBeans (к примеру, показывать назначение функции).

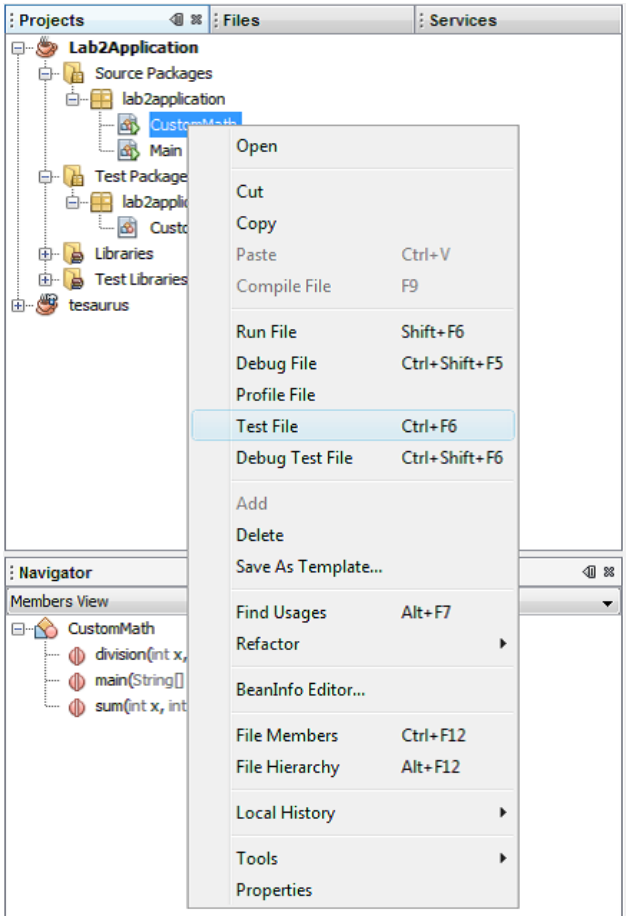
Созданный по умолчанию код класса тестов:

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package lab2application;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Alexander Sirenko
 */
public class CustomMathTest {

    public CustomMathTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        System.out.println("sum");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of division method, of class CustomMath.
     */
    @Test
    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);
        fail("The test case is a prototype.");
    }

    /**
     * Test of main method, of class CustomMath.
     */
    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        CustomMath.main(args);
        fail("The test case is a prototype.");
    }

}
```

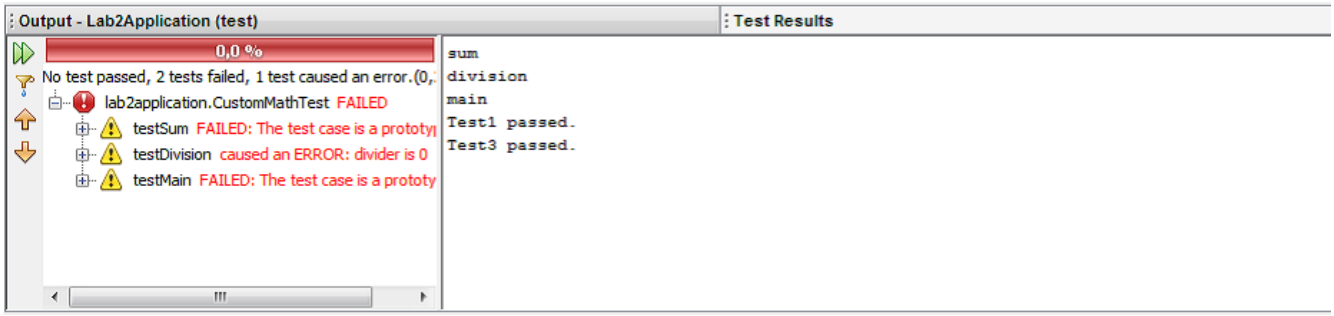
В коде тестов можно видеть аннотации: информация о назначении методов с символом @ (@BeforeClass, @AfterClass, @Test). Аннотация @Test отмечает методы, автоматически запускаемые средой тестирования. @BeforeClass и @AfterClass содержат действия, которые необходимо выполнить до запуска тестов класса (например, подключение к базе данных, или подготовку данных для обработки) или после выполнения тестов (например, отключение от базы данных, восстановление исходного ее состояния).

**Запуск созданных по умолчанию тестов**

Запуск тестов выполняется через контекстное меню тестируемого класса, пунктом Test File.




Вкладка Test Results отображает выводимые на консоль сообщения ( в данном случае размещенные нами в функции main проверки), а также результаты проверки методов тестируемого класса (3 неудачи).




Иерархия классов JUnit:

- java.lang.Object
  - org.junit.Assert
  - org.junit.Assume
  - java.lang.Throwable (implements java.io.Serializable)
    - java.lang.Error
      - java.lang.AssertionError
      - org.junit.ComparisonFailure
    - org.junit.Test.None

Annotation Type Hierarchy

- org.junit.Test (implements java.lang.annotation.Annotation)
- org.junit.Ignore (implements java.lang.annotation.Annotation)
- org.junit.BeforeClass (implements java.lang.annotation.Annotation)
- org.junit.Before (implements java.lang.annotation.Annotation)
- org.junit.AfterClass (implements java.lang.annotation.Annotation)
- org.junit.After (implements java.lang.annotation.Annotation)

Для проверки правильности выполнений метода в JUnit предусмотрена группа методов Assert, проверяющие условия и в случае несовпадения отмечающие тест не пройденным.

Функция fail() принудительно отмечает тест не пройденным. Используется, если мы реализуем некую проверку самостоятельно и она не отлавливается функцией assert.

**Игнорирование теста**

В некоторых ситуациях может понадобиться отключить некоторые тесты. Например, возможно в текущей версии используемой вами библиотеки имеется ошибка, или по какой-то причине определенный тест не может быть выполнен в текущей среде. D B JUnit 3.8.x, чтобы отключить тесты, приходилось их комментировать. B JUnit 4 для этих целей вам нужно просто промаркировать игнорируемый тест с помощью аннотации @Ignore.

Например

```
public class CalculatorTest {
                @Ignore("Not running ")
                @Test
                public void testTheWhatSoEverSpecialFunctionality() {
                }
}
```

Текст, который передается в аннотации @Ignore, поясняет причину пропуска теста и может использоваться средой разработки. Указывать текст не обязательно, но очень полезно всегда задавать сообщение для того, чтобы позже не забыть про то, что этот тест отключен.

**Обработка исключений**

Исключения могут быть правильным поведением метода при определенных условиях ( например, исключение отсутствия файла в случае, если он не доступен ). Можно обрабатывать исключение в тесте с помощью блока try…catch(), либо передавать его далее с помощью ключевого слова throws в описании метода. Изменим метод testDivision таким образом, чтобы он проверял корректное поведение при делении на 0. Корректным поведением в данном случае является генерация исключения.

```
    @Test
    public void testDivisionByZero() {
        int x = 0;
        int y = 0;
        int expResult = 0;
        try{
            int result = CustomMath.division(x, y);
            assertEquals(expResult, result);
            if(y==0) fail("Деление на ноль не создает исключительной ситуации");
        }catch(IllegalArgumentException e){
            if(y!=0) fail("Генерация исключения при ненулевом знаменателе");
        }
    }
```

**Параметризованные тесты**

Для проверки бизнес-логики приложений регулярно приходится создавать тесты, количество которых может существенно колебаться. В предшествующих версиях JUnit это приводило к значительным неудобствам - главным образом из-за того, что изменение групп параметров в тестируемом методе требовало написания отдельного тестового сценария для каждой группы. В версии JUnit 4 реализована возможность, позволяющая создавать общие тесты, в которые можно направлять различные значения параметров. В результате вы можете создать один тестовый сценарий и выполнить его несколько раз - по одному разу для каждого параметра.

Создание параметрического теста в JUnit 4 производится в пять шагов:

1. Создание типового теста без конкретных значений параметров.
2. Создание метода static, который возвращает тип Collection и маркирует его аннотацией @Parameter.
3. Создание полей класса для параметров, которые требуются для типового метода, описанного на шаге 1.
4. Создание конструктора, которые связывает параметры коллекции шага 2 с соответствующими полями класса, описанными на шаге 3.
5. Указание параметризованного запуска тестов с помощью класса Parametrized.

**Задание**

Частично реализовать разработанные в предыдущей лабораторной работе методы классов из UML-диаграмм информационной системы. Реализацию продумать таким образом, чтобы в методах присутствовали следующие варианты реализации функциональности:
- для порождающих паттернов (Фабрика, Строитель) - методы должны создавать не менее двух различных экземпляров классов
- для структурных паттернов (Фасад, Компоновщик) - методы должны реализовать генерацию не менее двух различных элементов управления информационной системы (кнопки, списки и т.д.)
- для паттернов поведения (Шаблонный метод, Посетитель) - методы должны содержать не менее двух вычислительных алгоритмов (например, решение уравнений, поиск максимума, сортировка).

Продумать какие параметры должны принимать и возвращать реализованные методы (с примерами)

На **удовлетворительную оценку** разработать Unit-тесты для паттернов поведения (2 теста)

На **хорошую оценку** разработать Unit-тесты для паттернов поведения, порождающих паттернов (4 теста).

На **отличную оценку** разработать Unit-тесты для паттернов поведения, структурных паттернов и порождающих паттернов (6 тестов, причем (!) хотя бы один из них **должен быть параметризованным**).

Для сдачи лабораторной работы подготовить UML-диаграммы с **выделенными** реализованными методами, отдельно приготовить листинги методов и соответствующих им тестов.

Продемонстрировать на компьютере работу разработанных тестов (**как корректное прохождение тестов, так и ошибки**).