Национальный Исследовательский Университет
«Московский Энергетический Институт»
Кафедра прикладной математики и искусственного интеллекта

Тема: Модульное тестирование кода на языке Java с применением JUnit.

Студент: Ростовых Александра

Москва 2021

Цель работы

Научиться разрабатывать модульные тесты для кода на языке Java с применением JUnit. Разработать модульный тест с применением библиотеки JUnit 4 для программного кода, разработанного на языке Java.

1. Создать в среде разработки Eclipse (или IntelliJ IDEA) консольный проект Java (Java Application). Подключить к проекту библиотеку JUnit версии 4.

В IDE IntelliJ IDEA создаем консольный проект, система сборки – Gradle.

2. Создать несколько классов, которые будут имитировать тестируемую логику.

Класс Person:

```
package Classes;

public class Person {
    public int age;
    public String name;
    public double weight, height;
    double def = 50;
    double defh = 165;
    public Person(String str, int age1,double w, double height1)
    {
        this.name = str;
        this.age = age1;
        this.weight = w;
        this.height = height1;

    }
    public Person()
    {
        this.name = "Ростовых";
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }
    public double IMT(double w, double height1)
    {
        if (w > 0 && height1>0)
        {
            return w/(height1*height1/10000);
        }
        else return 0;
    }

    public double IMT()
    {
        return (weight > 0 && height > 0) ? (weight / (height * height /
10000)) : 0;

    }
    public int category( int age1, double w, double height1)
    {
        if (age1>0)
        {
            if (w > 0 && height > 0)
```

```
                return (int)Math.abs(age1 - IMT(w, height1))+1;
            else return (int)(IMT(def, defh)/5)+2;
        }
        else return -1;
    }

    public boolean IsCorrect()
    {
        return (name == "" || age < 0 || weight < 0 || height < 0) ? false :
true;
    }
    public void FullName() throws PersonException {
        if (this.name == "")
        {
            throw new PersonException("Name can't be empty");
        }
        else
        {
            this.name = "Ростовых Александра Дмитриевна";
        }

    }

}
```

И класс исключений:

```
package Classes;

public class PersonException extends Exception {
    public PersonException(String msg)  {
        super(msg);
    }
}
```

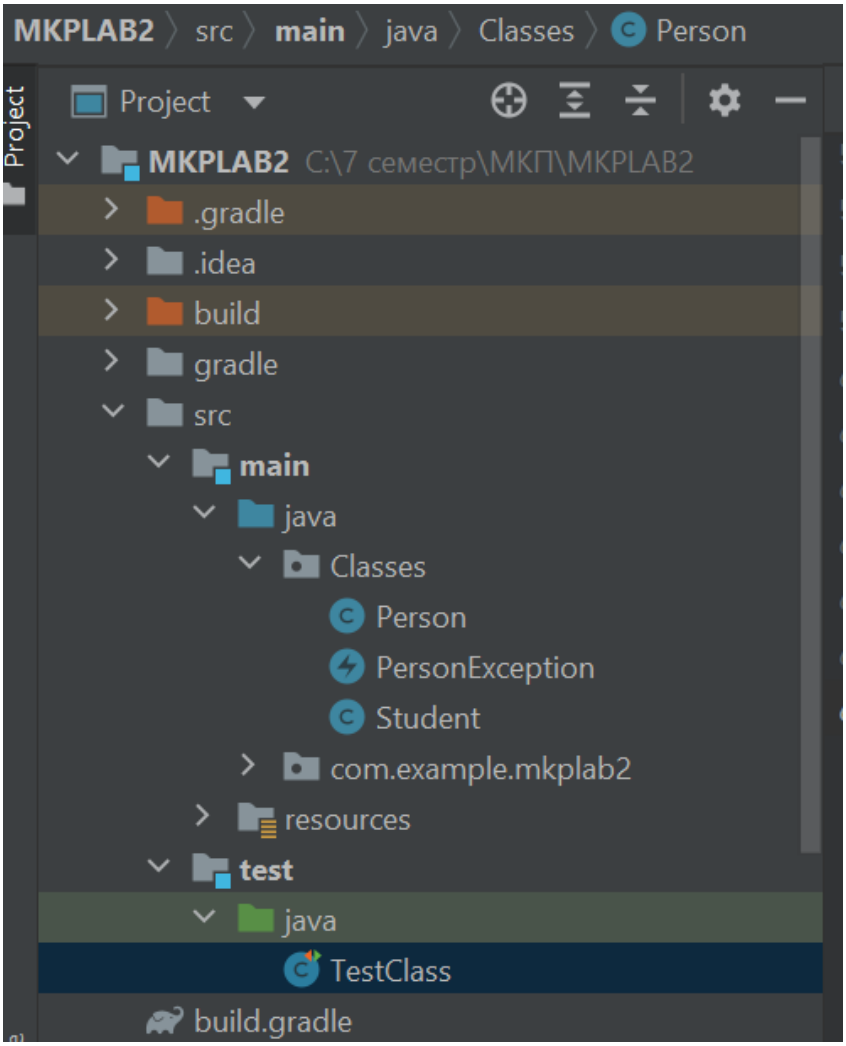
А так же класс Student:

```
package Classes;

public class Student
{
    public String name;
    public int score;
    public int Sum;

    public Student(String name, int score)

    {
        this.name = name;
        this.score = score;
        this.Sum = score;
    }
    public String getName() {
        return this.name;
    }

    public double getScore() {
        return this.score;
    }
```

```
    public void Newscore(int score)
    {
        this.score += score;
        this.Sum += score;
    }
    public double averagescore(int Sum, int count)
    {
        return (double)Sum / count;
    }

    public int ID(int Sum, int count)
    {
        return (int)(averagescore(Sum, count)/5+this.score*234-this.Sum);
    }
}
```

3. Создать в проекте новый тестовый класс.

Создали новый тестовый класс, назвали его TestClass:



4. Описать в этом классе функции setUp() и tearDown() и отметить их аннотациями @Before, @After.

Аннотация Before обозначает методы, которые будут вызваны до исполнения теста, методы должны быть public void. Традиционно метод называют setUp.

В лабораторной используется более новая версия – JUnit 5, в которой аналогом Before является BeforeEach

Аннотация After обозначает методы, которые будут вызваны после выполнения теста, методы должны быть public void. Традиционно метод называют tearDown.

В лабораторной используется более новая версия – JUnit 5, в которой аналогом After является AfterEach.

```
private Person person;

@BeforeEach
void setUp() {
    this.person = new Person("Rostovykh", 20, 50, 156);
}

@AfterEach
void tearDown() {
    this.person = null;
}
```

5. Разработать не менее пяти тестирующих функций, отметить их соответствующими аннотациями. При разработке этих функций следует активно применять функции assertEquals, assertTrue, assertFalse.

```
@Test
void TestMethod1()
{
    Assertions.assertEquals(20.0, this.person.IMT(),1.0);
    Assertions.assertTrue(this.person.IsCorrect());
    Assertions.assertNotNull(this.person.name);
}
@Test
void TestMethod2()
{
    Person person2 = new Person("", 21, 54, 172);
    Assertions.assertEquals(25.0, person2.IMT(100,200) );
    Assertions.assertFalse(person2.IsCorrect());
    Assertions.assertNotEquals(0.0, this.person.IMT()+2);
}
@Test
void TestMethod3() throws PersonException {
    Person person2 = new Person("", 17, 49, 163);
    Throwable thrown = assertThrows(PersonException.class, () -> {
        person2.FullName();
    });
    Assertions.assertNotNull(thrown.getMessage());
    Assertions.assertEquals(1, this.person.category(18, 50, 170));
}

@Test
```

```
void TestMethod4() throws PersonException {
    this.person.FullName();
    Assertions.assertEquals(9, this.person.getNum());
}

@Test
void TestMethod5() {
    Assertions.assertNotNull(this.person.getName());
    Assertions.assertEquals(20, this.person.getAge());
    Person person2 = new Person(null, 17, 49, 163);
    Assertions.assertNull(person2.getName());
}
@Test
void TestMethod6() {
    Student student = new Student("Rostovykh", 20);
    Assertions.assertEquals(5.0, student.averagescore(student.Sum,4));
    Assertions.assertNotNull(student.getName());
    Assertions.assertNotNull(student.getScore());
    Assertions.assertEquals(student.getScore(), this.person.getAge());
}

@Test
void TestMethod7() {
    int count = 9;
    Student student = new Student("Alexandra", 42);
    Assertions.assertFalse(4.5 == student.averagescore(student.Sum, count));
    student.Newscore(3);
    count += 1;
    Assertions.assertTrue(4.5 == student.averagescore(student.Sum, count));
    Assertions.assertNotNull(student.getScore());
}

@ParameterizedTest
@ValueSource(ints = { 0, 5, 14, 32, 15, 3})
void parameterTest(int Score) {
    Student student = new Student("Rostovykh", 21);
    student.Newscore(Score);
    int expected=21+Score;
    Assertions.assertEquals(expected, student.getScore());
}
```

6. Добавить в тестовые методы спецификацию ожидаемых исключений.

В JUnit5:

```
@Test
void TestMethod3() throws PersonException {
    Person person2 = new Person("", 17, 49, 163);
    Throwable thrown = assertThrows(PersonException.class, () -> {
        person2.FullName();
    });
    Assertions.assertNotNull(thrown.getMessage());
    Assertions.assertEquals(1, this.person.category(18, 50, 170));
}

@Test
void TestMethod4() throws PersonException {
    this.person.FullName();
    Assertions.assertEquals(9, this.person.getNum());
}
```

В JUnit4 самый простой способ сообщить тестовому фреймворку о том, что ожидается исключение – указать дополнительный параметр expected в аннотации @Test:

```
@Test(expected = PersonException.class)
```

Но этот способ имеет некоторые недостатки:

- Нельзя проверить текст сообщения или другие свойства возникшего исключения.
- Нельзя понять, где именно возникло исключение

Поэтому стоило бы использовать try-catch:

```
@Test
void TestMethod3() throws PersonException {
    Person person2 = new Person("", 17, 49, 163);
    try {
        person2.FullName();
    } catch (PersonException thrown) {
        Assertions.assertNotNull(thrown.getMessage());
    }
    Assertions.assertEquals(1, this.person.category(18, 50, 170));
}
```

7. Разработать параметризованный тест.

Это позволяет нам многократно выполнять один метод тестирования с разными параметрами.

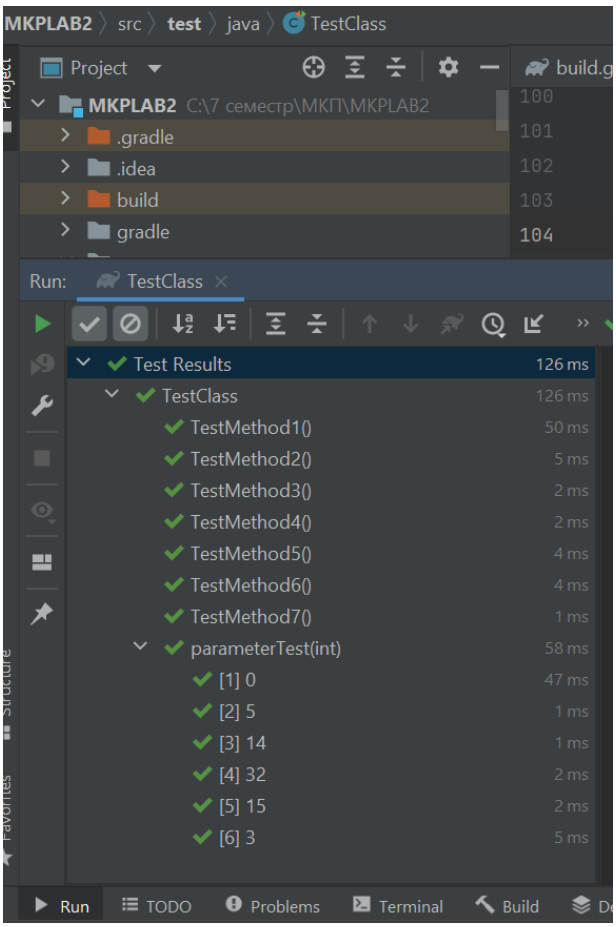
Параметризованные тесты похожи на другие тесты, за исключением того, что мы добавляем аннотацию @ParameterizedTest

```
@ParameterizedTest
@ValueSource(ints = { 0, 5, 14, 32, 15, 3})
void parameterTest(int Score) {
    Student student = new Student("Rostovykh", 21);
    student.Newscore(Score);
    int expected=21+Score;
    Assertions.assertEquals(expected, student.getScore());
}
```

Каждый раз он присваивается другое значение из массива @ValueSource параметру числового метода.

8. Запустить проверку разработанного тестового класса.

9. Скомпилировать и запустить проект. Посмотреть на результат теста.



Все тесты пройдены успешно!

10. Внести в тестируемые классы изменения, приводящие к ошибкам.

11. Скомпилировать и запустить проект. Посмотреть, пойманы ли ошибки модульным тестом.

См.ниже

12. Добавить к одному из тестовых методов спецификацию ожидаемого времени работы

```
@Test
@Timeout(value=90, unit= TimeUnit.MILLISECONDS)
void TestTime() throws InterruptedException {
    Student student = new Student("Rostovykh", 21);
    student.Newscore(10);
    int expected=21+10;
    Assertions.assertEquals(expected, student.getScore());
}
```

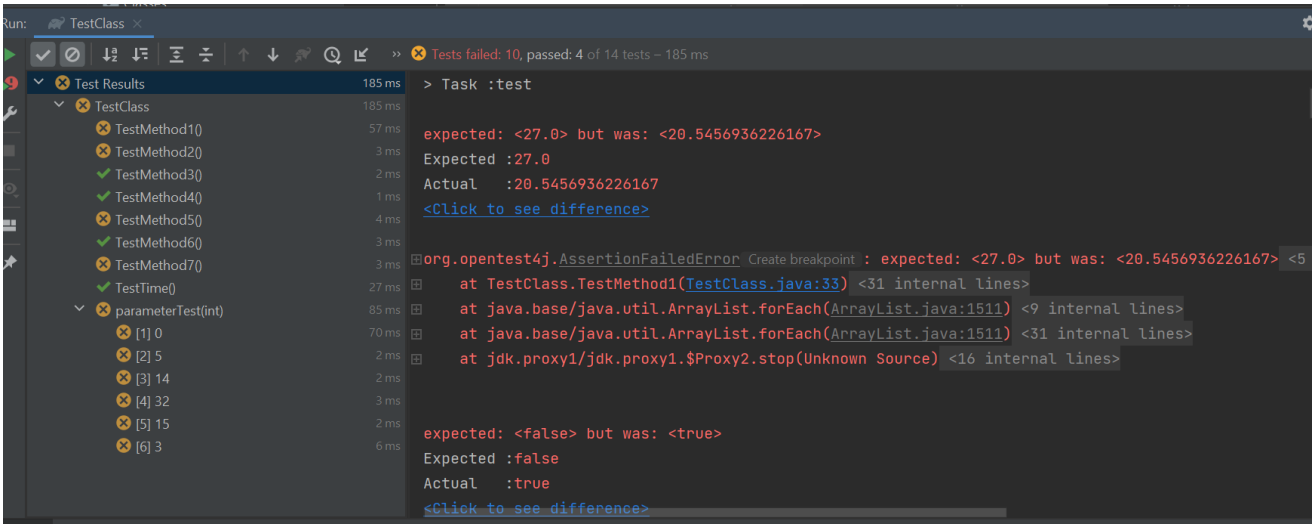
Варианты использования. Отсутствие единицы измерения эквивалентно использованию секунд.

10	@Timeout(10)

10 ns	@Timeout(value = 10, unit = NANOSECONDS)
10 µs	@Timeout(value = 10, unit = MICROSECONDS)
10 ms	@Timeout(value = 10, unit = MILLISECONDS)
10 s	@Timeout(value = 10, unit = SECONDS)
10 m	@Timeout(value = 10, unit = MINUTES)
10 h	@Timeout(value = 10, unit = HOURS)
10 d	@Timeout(value = 10, unit = DAYS)

В JUnit4: @Test (timeout = 1000) 1000- время в миллисекундах

Вносим ошибки и проверяем, пойманы ли они тестами.



Ошибки пойманы, среда выводит дополнительную информацию, так же видим, что тест со спецификацией ожидаемого времени работы пройден.