СБЕРБАНК ТЕХНОЛОГИИ

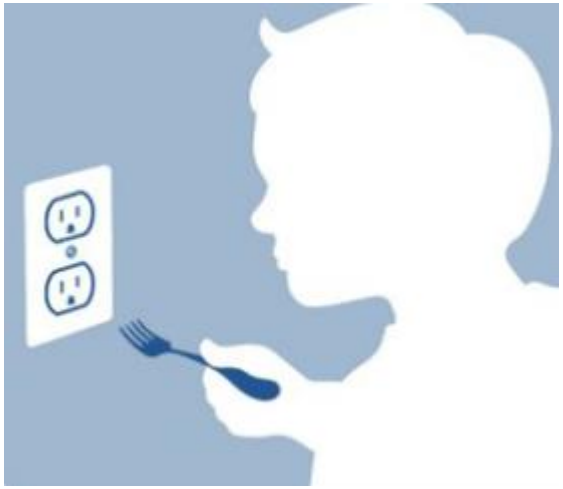
# МОДУЛЬНОЕ ТЕСТИРОВАНИЕ


Дата

# СОДЕРЖАНИЕ

# ЗАЧЕМ?

# МОДУЛЬНОЕ ТЕСТИРОВАНИЕ



*Модуль* – наименьший компилируемый участок программы.

*Модульное (unit) тестирование* – изолированная проверка отдельных модулей программы путём запуска тестов в искусственной среде. Тесты должны быть повторяемыми и пишутся для каждой нетривиальной функции модуля.

# ПРЕИМУЩЕСТВА

- **Спокойный рефакторинг.**
  Модуль по-прежнему работает корректно.

- **Простая отладка.**
  Достаточно использовать отдельный тест.

- **Упрощение интеграции.**
  Нет сомнений по поводу корректности отдельных модулей, а значит можно тестировать «снизу вверх»: сначала отдельные модули, потом программу в целом.

- **Документирование кода.**
  Модульный тест – «живой документ» тестируемого модуля.

- **Отделение интерфейса от реализации.**
  Тест не должен выходить за границу модуля. Это заставляет разработчика абстрагироваться от конкретных реализаций сторонних для теста классов. Результат - минимум зависимостей.

# НЕДОСТАТКИ

- **Много дополнительной работы.**
  Да, это так. Но существует огромное количество фрэймворков, которые позволяют существенно упростить эту работу.

- **Лишняя трата времени на выполнение этой дополнительной работы.**
  А вот это не так! Тесты сокращают (иногда сильно) длительность поиска ошибок и отладки кода на большее количество времени чем то, что было затрачено на их написание.

# ПОПУЛЯРНЫЕ ФРЭЙМВОРКИ

- **JUnit**

  Самый популярный, является де факто стандартом написания модульных тестов.

- **TestNG**

  Аналог JUnit.

- **Mockito**

  Простое написание фиктивных (mock) реализаций.

- **EasyMock**

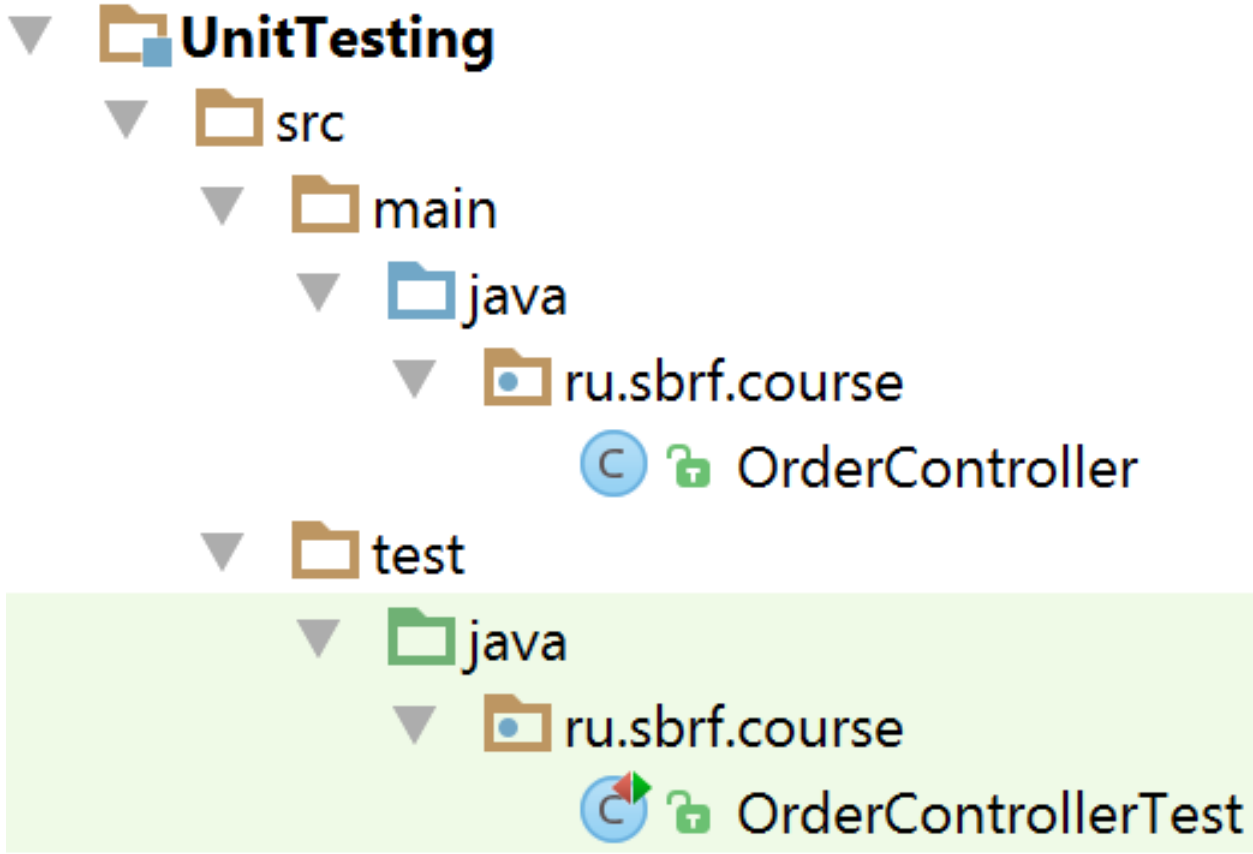
  Аналог Mockito.

- **PowerMock**

  Добавляет mock библиотекам (Mockito и EasyMock) новые возможности.

# JUnit ШАГ №1

Создайте класс, где будут размещены отдельные тесты.



# JUnit ШАГ №2

Определите отдельные тесты.

- Какой участок кода (метод) будет тестироваться?
- Каковы предусловия?
- Входные параметры.
- Ожидаемые результаты.

# JUnit ШАГ №3

Добавьте методы по созданию исходного для теста/тестов состояния (например, инициализация ресурсов, создание необходимых объектов).

Используйте следующие аннотации:

- для всех тестов модуля
  - @BeforeClass
- для каждого теста в отдельности
  - @Before
  - @Rule

# JUnit ШАГ №3

```
public class SomeTest {

    private File output;

    @Rule
    public TemporaryFolder tempFolder = new TemporaryFolder();

    @BeforeClass
    public static void setUpClass() {
        ...
    }

    @Before
    public void setUp() {
        output = new File("1.txt");
    }

    …
}
```

Добавьте сами тесты, где вызовите проверяемый участок кода.

Используйте аннотацию @Test:

- @Test – проверяемый метод не должен бросить исключение.
- @Test(expected = SomeException.class) – проверяемый метод должен бросить исключение SomeException.
- @Test(timeout = 100) – тест провалится, если на его исполнение потратится более 100 миллисекунд.

Помните:

- один тест на одну проверяемую ситуацию;
- тесты не должны зависеть от выполнения других тестов, таким образом порядок исполнения тестов не должен имеет значения.

```
public class OrderControllerTest {

    @Test(expected = NullPointerException.class)
    public void testShouldThrowNpeIfItemIsNull() {
        orderController.getItemDiscountForClient(client, null);
    }

    @Test
    public void testDiscountForClientMinimum15IfPrivileged() {
        …
        int discount = orderController.getItemDiscountForClient(client,
item);
        …
    }

}
```

# JUnit ШАГ №5

Добавьте код по оценке результата исполнения проверяемого метода, для чего используйте статические методы класса Assert:

- assertTrue(String message, **boolean** condition)
- assertFalse(String message, **boolean** condition)

- assertEquals(String message, Object expected, Object actual)
- assertNotEquals(String message, Object unexpected, Object actual)

- assertArrayEquals(String message, Object[] expecteds, Object[] actuals)

- assertNull(String message, Object object)
- assertNotNull(String message, Object object)

- assertSame(String message, Object expected, Object actual)
- assertNotSame(String message, Object unexpected, Object actual)

- assertThat(String reason, T actual, Matcher<? **super** T> matcher)

# JUnit ШАГ №5

```
public class OrderControllerTest {

    @Test
    public void testDiscountForClientMinimum15IfPrivileged() {
        Client client = newPrivilegedClient();
        Item item = new Item();

        int discount = orderController.getItemDiscountForClient(client,
item);

        assertTrue("Discount for privileged clients should not be less
than 15%", discount >= 15);
    }

}
```

Добавьте код, возвращающий систему в то состояние, в котором она была до исполнения теста.

Используйте следующие аннотации:

- для всех тестов модуля
  - @AfterClass
- для каждого теста в отдельности
  - @After
  - @Rule

# JUnit ШАГ №6

```
public class SomeTest {

    private File output;

    …

    @AfterClass
    public static void tearDownClass() {
        …
    }

    @After
    public void tearDown() {
        output.delete();
    }

}
```

# JUnit ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

1. Добавьте аннотацию @RunWith(Parameterized.class) к классу теста.

2. Определите набор параметров с ожидаемыми значениями: public static метод с аннотацией @Parameterized.Parameters, возвращающий Iterable<Object[]>. Один класс теста на один такой набор.

3. Определите public члены класса теста с аннотацией @Parameterized.Parameter. Аргумент аннотации – порядковый номер элемента в наборе. Используйте эти члены в качестве значений для теста.

# JUnit ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

```
@RunWith(Parameterized.class)
public class CalculatorTest {
    @Parameter
    public int a;

    @Parameter(1)
    public int b;

    @Parameter(2)
    public int expected;

    private Calculator calculator = new Calculator();

    @Parameters
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][]{{5, 2, 7}, {1, 1, 2}, {2, 1, 3}});
    }

    @Test
    public void testAdd() throws Exception {
        assertEquals(expected, calculator.add(a, b));
    }
}
```

# JUnit ОТКЛЮЧЕНИЕ ТЕСТОВ

- Для статического отключения тестов используйте аннотацию @Ignore или @Ignore("Why disabled").

- Если тест необходимо отключать на основании какого-то условия, то используйте статические методы класса Assume:

  - assumeTrue(String message, **boolean** b)
  - assumeFalse(String message, **boolean** b)

  - assumeNotNull(Object... objects)

  - assumeThat(String message, T actual, Matcher<T> matcher)

  - assumeNoException(String message, Throwable e)

# JUnit ОТКЛЮЧЕНИЕ ТЕСТОВ

```
public class OrderControllerTest {

    @Test
    @Ignore
    public void testDiscountForClientMinimum15IfPrivileged() {
        …
    }

    @Test
    public void testAlwaysCheckItemDiscount() {
        String osName = System.getProperty("os.name");
        assumeTrue(osName.contains("Windows"));
        …
    }

}
```

# JUnit НАБОРЫ И КАТЕГОРИИ

- Тесты можно объединять в наборы (suites). Это позволит выполнять набор тестов целиком. Для этого используйте аннотации @RunWith(Suite.class) и @Suite.SuiteClasses.

- Используйте категории для того, чтобы из всего набора тестовых классов и методов выполнять только те, что указаны в аннотации @IncludeCategory.

# JUnit НАБОРЫ И КАТЕГОРИИ

```
public interface SlowTests {
}

public interface FastTests {
}

public class OrderControllerTest {
    @Test
    public void testDiscountForClientMinimum15IfPrivileged() {
        …
    }

    @Category(SlowTests.class)
    @Test
    public void testAlwaysCheckItemDiscount() {
        …
    }
}

@Category({SlowTests.class, FastTests.class})
public class CalculatorTest {
    …
}
```

# JUnit НАБОРЫ И КАТЕГОРИИ

```
@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@SuiteClasses({CalculatorTest.class, OrderControllerTest.class})
public class SlowTestsSuite {
    // OrderControllerTest.testAlwaysCheckItemDiscount + all tests in
CalculatorTest
}

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@ExcludeCategory(FastTests.class)
@SuiteClasses({CalculatorTest.class, OrderControllerTest.class})
public class FastTestsSuite {
    // OrderControllerTest.testAlwaysCheckItemDiscount + but not tests in
CalculatorTest
}
```

# MOCKING?

*Mock-объект* («объект имитация») представляет собой фиктивную реализацию интерфейса, предназначенную исключительно для тестирования взаимодействия с ним.

# @Mock, mock

Для создания mock'а используйте аннотацию @Mock:

```
public class SomeControllerTest {
    @Mock
    private SomeRepository mockedRepository;
}
```

Или статический метод Mockito.mock:

```
public class SomeControllerTest {
    private SomeRepository mockedRepository;

    @Before
    public void setUp() {
        mockedRepository = mock(SomeRepository.class);
    }
}
```


# @Mock, mock

Для создания mock'а используйте аннотацию @Mock:

```
public class OrderControllerTest {
    @Mock
    private DiscountRegistry mockedDiscountRegistry;
}
```

Или статический метод Mockito.mock:

```
public class OrderControllerTest {
    private DiscountRegistry mockedDiscountRegistry;
}

public class SomeControllerTest {
    private SomeRepository mockedRepository;

    @Before
    public void setUp() {
        mockedDiscountRegistry = mock(DiscountRegistry.class);
    }
}
```


# ИНИЦИАЛИЗАЦИЯ @Mock

Инициализируйте mock'и с аннотацией @Mock

Добавьте аннотацию @RunWith(MockitoJUnitRunner.class):

```
@RunWith(MockitoJUnitRunner.class)
public class OrderControllerTest {
    @Mock
    private DiscountRegistry mockedDiscountRegistry;
}
```

Или вызовите статический метод MockitoAnnotations.initMocks:

```
public class OrderControllerTest {
    @Mock
    private DiscountRegistry mockedDiscountRegistry;

    @Before
    public void setUp() {
        initMocks(this);
    }
}
```

Используйте verify, чтобы проверить факт вызова метода.

```
@Test
public void testList() {
    // Using mock object
    mockedList.add("One");
    mockedList.add("Two");
    mockedList.add("Two");

    // Verification
    verify(mockedList).add("One");
    verify(mockedList, times(2)).add("Two");
    verify(mockedList, never()).add("Three");
    verify(mockedList, atLeast(3)).add(anyString());
}
```

# when, thenReturn, doReturn

Используйте when+thenReturn или doReturn+when, чтобы подменять возвращаемое методом значение.

```
@Test
public void testList() {
    when(mockedList.get(anyInt())).thenReturn("Four");
    when(mockedList.get(5)).thenReturn("Five");
    doReturn("Six").when(mockedList).get(100);

    assertEquals("Four", mockedList.get(500));
    assertEquals("Five", mockedList.get(5));
    assertEquals("Six", mockedList.get(100));
}
```

# doThrow, thenThrow

Используйте doThrow/thenThrow, чтобы эмулировать бросание исключения.

```
@Test(expected = NullPointerException.class)
public void testShouldThrowNullPointerException() {
    doThrow(NullPointerException.class).when(mockedList).clear();

    mockedList.clear();
}

@Test(expected = IllegalStateException.class)
public void testShouldThrowIllegalStateException() {
    when(mockedList.get(anyInt()))
            .thenThrow(new IllegalStateException());

    mockedList.get(0);
}
```

Для проверки порядка вызова методов используйте inOrder.

```
@Mock
private List<String> mockedList1;

@Mock
private List<String> mockedList2;

@Test
public void testList() {
    mockedList1.add("Called first");
    mockedList2.add("Called second");

    InOrder inOrder = inOrder(mockedList1, mockedList2);

    inOrder.verify(mockedList1).add("Called first");
    inOrder.verify(mockedList2).add("Called second");
}
```

# verifyZeroInteractions

Необходимо проверить, что не было ни одного обращения к конкретному mock'у? – Используйте verifyZeroInteractions.

```
@Mock
private List<String> mockedList;

@Test
public void testList() {
    // mockedList.get(0);
}

@After
public void after() {
    verifyZeroInteractions(mockedList);
}
```

# @Spy, spy

Можно mock'ать и готовые реализации.

Используйте аннотацию @Spy:

```
public class ListTest {
    @Spy
    private List<String> spyOnList = new ArrayList<>(100);

    @Spy
    private HashSet<String> spyOnSet;
}
```

Или статический метод Mockito.spy:

```
public class ListTest {
    private List<String> spyOnList = spy(new ArrayList<String>());
}
```

# @Spy, spy

Verify, when и прочие также работают.

```
@Test
public void testListVerify() {
    spyOnList.add("One");

    verify(spyOnList).add("One");
    verifyNoMoreInteractions(spyOnList);
}

@Test
public void testListWhen() {
    when(spyOnList.contains("One")).thenReturn(false);

    spyOnList.add("One");

    assertEquals("One", spyOnList.get(0));
    assertFalse(spyOnList.contains("One"));
}
```


# ВОЗМОЖНОСТИ

PowerMock расширяет возможности EasyMock и Mockito, предоставляя возможность создания моков для:

- статических методов;
- финальных методов и классов;
- закрытых (private) методов;
- new, т.е. конструкторов классов.


# СТАТИЧЕСКИЕ МЕТОДЫ

1. Добавьте к классу теста аннотацию @RunWith(PowerMockRunner.class).

2. Добавьте к классу теста аннотацию @PrepareForTest аргументом которой является класс, содержащий статический метод.

3. Используйте PowerMock.mockStatic(), чтобы замокать весь класс целиком, или PowerMockito.spy(class), чтобы замокать конкретный метод.

4. Используйте Mockito.when() для установки ожидаемых значений.

# СТАТИЧЕСКИЕ МЕТОДЫ

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(NumberGenerator.class)
public class SomeClassTest {
    @Test
    public void testStaticGenerateLong() {
        PowerMockito.mockStatic(NumberGenerator.class);
        Mockito.when(NumberGenerator.generateLong())
                .thenReturn(100L).thenReturn(200L);

        long generatedValue1 = NumberGenerator.generateLong();
        long generatedValue2 = NumberGenerator.generateLong();

        Assert.assertEquals(100L, generatedValue1);
        Assert.assertEquals(200L, generatedValue2);

        PowerMockito.verifyStatic(Mockito.times(2));
        NumberGenerator.generateLong();
    }
}
```

# ФИНАЛЬНЫЕ МЕТОДЫ

```
public final class Key {
    public long getId() {
        return 100L;
    }
}

@RunWith(PowerMockRunner.class)
@PrepareForTest(Key.class)
public class SomeClassTest {
    @Test
    public void testFinalGetId() {
        Key key = PowerMockito.spy(new Key());
        Mockito.when(key.getId()).thenReturn(123L);

        long keyId = key.getId();

        Assert.assertEquals(123, keyId);
    }
}
```

# ИСПОЛЬЗУЕМЫЕ МАТЕРИАЛЫ

- http://junit.org/junit4/
- http://mockito.org/
- https://github.com/jayway/powermock

СПАСИБО!