Test Frameworks

Voronezh, 2018

T · · Systems

Agenda

- UNIT и INTEGRATION тесты
- Приемущества модульного тестирования (UNIT тестов)
- JUnit - библиотека для модульного тестирования
- Mock-библиотеки
- Разработка через тестирование TDD (test-driven development)

If Not WE, Then Who?

- **Кто должен писать юнит-тесты?**

Тестировщик

Программист

T · · Systems

UNIT и INTEGRATION тесты

- **Unit Test:**
  - Процесс в программировании, позволяющий **проверить** на корректность **отдельные модули** исходного кода **программы**.
  - Идея состоит в том, чтобы писать **тесты для каждого нетривиального метода**.

- **Integration Test:**
  - Одна из **фаз тестирования** программного обеспечения, **при котором** отдельные программные **модули объединяются и тестируются в группе**.

2 unit tests. 0 integration tests…

T Systems

Приемущества UNIT тестов

- **Поощрение изменений**

  Модульное тестирование **позже позволяет** программистам **проводить рефакторинг**, **будучи уверенными**, **что модуль по-прежнему работает корректно**.

- **Упрощение интеграции**

  Модульное 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**

  Модульные тесты можно рассматривать как "живой документ" для тестируемого класса. **Клиенты**, которые не знают, как использовать данный класс, **могут использовать юнит-тест в качестве примера**.

- **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**.

Ограничения UNIT тестов

- **Ошибки интеграции системного уровня:**
  - Происходит **тестирование каждого из модулей по отдельности**.
  - Это означает, что **ошибки интеграции**, **системного уровня, функций**, исполняемых в нескольких модулях **не будут определены**.

- **Производительность:**
  - Данная технология **бесполезна для** проведения **тестов на производительность**.

- **Таким образом**, модульное тестирование более **эффективно при использовании в сочетании с другими методиками тестирования**.

T · · Systems

JUnit – библиотека для модульного тестирования

- **JUnit** принадлежит семье фреймворков **xUnit** для разных языков программирования.

- **JUnit** породил систему расширений - **JMock**, **EasyMock**, **DbUnit**, **HttpUnit**

Функциональность JUnit 4 (1)

- JUnit 4 полностью **построен на аннотациях**.

- Все, что вам нужно сделать – это обозначить тестовый метод с помощью аннотации:

  @Test

- Настройка тестов:

  @BeforeClass

  @AfterClass

  @Before

  @After

- Тестирование исключений:

  @Test(expected=AnyException.class)

T · · Systems

Функциональность JUnit 4 (2)

- TimeOut:

  @Test(timeout=5000)

- Игнорирование тестов:

  @Ignore("Not running because <fill in a good reason here>")

- Наборы тестов:

  @RunWith(value=Suite.class)
  @SuiteClasses(value= { Test1.class, Test2.class })

- Параметризированные тесты:

  @RunWith(value=Parameterized.class)

T · · Systems

JUnit4 в примерах (1)

```
public class Calculator {

    public Integer maxValue(Integer a, Integer b) {

        if (a == null || b == null) {
            throw new IllegalArgumentException("Param must be not null!");
        }

        return a > b ? a : b;
    }

}
```

JUnit4 в примерах (2)

```
public class CalculatorTest  {
        private Calculator calculator;

        @Before
        public void initCalculator() {
                calculator = new Calculator();
        }

        @Test
        public void maxValueTest() {
                Integer actual = calculator.maxValue(10, 20);
                Assert.assertEquals(20, actual);
        }

        @Test(expected = IllegalArgumentException.class)
        public void exceptionTest() {
                calculator.maxValue(10, null);
        }
}
```

T · · Systems

JUnit4 дополнительные возможности (1)

- Правила (**@Rule**)
  - Временные файлы:

    @Rule public final **TemporaryFolder** folder = new TemporaryFolder();
  - Для задания таймаута:

    @Rule public final **Timeout** timeout = new Timeout(1000);
  - Для исключений:

    @Rule public final **ExpectedException** thrown = ExpectedException.none();
  - Правила - это некое **подобие утилит для тестов**, которые **добавляют функционал до и после выполнения теста**.

JUnit4 дополнительные возможности (2)

- Runner (**@RunWith**)
  - **JUnit4** - по умолчанию предназначена для запуска JUnit 4 тестов.
  - **Suite** - для запуска последовательности тестов. Для настройки запускаемых тестов используется аннотация **@SuiteClasses**.

- Categories (**@Category**) – организация тестов в категории.

JUnit4 дополнительные возможности (3)

- **Multithreaded** unit tests:

```
public interface Counter {
    int incrementAndGet();
}

public class DumbCounter implements Counter {

    private int counter;

    @Override
    public int incrementAndGet() {
        return ++counter;
     }
}
```

JUnit4 дополнительные возможности (4)

- **Multithreaded** unit tests:

```
public class AtomicIntCounter implements Counter {

    private AtomicInteger atomicInt = new AtomicInteger();

    @Override
    public int incrementAndGet() {
        return atomicInt.incrementAndGet();
    }
}
```

TestNG аналог JUnit (1)

- Фреймворк для тестирования **TestNG**, аналог **Junit**.

- Иерархия (http://habrahabr.ru/post/121234):

```
+- suite/
  +- test0/
  |  +- class0/
  |  |  +- method0(integration group)/
  |  |  +- method1(functional group)/
  |  |  +- method2/
  |  +- class1
  |     +- method3(optional group)/
  +- test1/
     +- class3(optional group, integration group)/
        +- method4/
```

TestNG аналог JUnit (2)

```
+- before suite/
  +- before group/
    +- before test/
      +- before class/
        +- before method/
          +- test/
        +- after method/
        ...
      +- after class/
      ...
    +- after test/
    ...
  +- after group/
  ...
+- after suite/
```

Функциональность TestNG (1)

- Аннотация **@Test** обозначает сами тесты.
  - Здесь размещаются проверки. Также применима к классам.

- Аннотации **@BeforeSuite, @AfterSuite** обозначают **методы, которые исполняются единожды до/после исполнения всех тестов**.
  - Здесь удобно располагать какие-либо тяжелые настройки общие для всех тестов, например, здесь можно создать пул соединений с базой данных.

- Аннотации **@BeforeClass, @AfterClass** обозначают **методы, которые исполняются единожды до/после исполнения всех тестов в классе**, идентичны предыдущим, но применимы к тест-классам.
  - Наиболее применим для тестирования какого-то определенного сервиса, который не меняет свое состояние в результате теста.

Функциональность TestNG (2)

- Аннотации **@BeforeTest, @AfterTest** обозначают **методы, которые исполняются единожды до/после исполнения теста** (тот, который включает в себя тестовые классы, не путать с тестовыми методами).
  - Здесь можно хранить настройки какой-либо группы взаимосвязанных сервисов, либо одного сервиса, если он тестируется несколькими тест-классами.

- Аннотации **@BeforeMethod, @AfterMethod** обозначают **методы, которые исполняются каждый раз до/после исполнения тестового метода**.
  - Здесь удобно хранить настройки для определенного бина или сервиса, если он не меняет свое состояние в результате теста.

- Аннотации **@BeforeGroups, @AfterGroups** обозначает **методы, которые исполняются до/после первого/последнего теста, принадлежащего к заданным группам**.

Функциональность TestNG (3)

- У всех этих аннотаций есть следующие параметры:
  - **enabled** – можно временно отключить, установив значение в false
  - **groups** – обозначает, для каких групп будет исполнен
  - **inheritGroups** – если true (а по умолчанию именно так), метод будет наследовать группы от тест-класса
  - **timeOut** – время, после которого метод "свалится" и потянет за собой все зависимые от него тесты
  - **description** – название, используемое в отчете
  - **dependsOnMethods** – методы, от которых зависит, сначала будут выполнены они, а затем данный метод
  - **dependsOnGroups** – группы, от которых зависит
  - **alwaysRun** – если установить в true, будет вызываться всегда независимо от того, к каким группам принадлежит, не применим к @BeforeGroups, @AfterGroups

JUnit vs. TestNG

Functionality - JUnit 4 vs TestNG

	Annotation Support	Exception Test	Ignore Test	Timeout Test	Suite Test	Group Test	Parameterized (primitive value)	Parameterized (object)	Dependency Test
TestNG	✓	✓	✓	✓	✓	✓	✓	✓	✓
JUnit 4	✓	✓	✓	✓	✓	✗	✓	✗	✗

- *"After go thought all the features comparison, I **suggest to use TestNG as core unit test framework** for Java project, because **TestNG** is **more advance in parameterize testing**, **dependency testing** and **suite testing** (Grouping concept). TestNG is meant for **high-level testing** and **complex integration test**. Its flexibility is especially useful with **large test suites**. In addition, TestNG also **cover the entire core JUnit4 functionality**. It's just no reason for me to use JUnit anymore."*

  http://www.mkyong.com/unittest/junit-4-vs-testng-comparison

T · · Systems

Mock Object (1)

- **Mock-объект** (от англ. *mock object*, буквально: объект-пародия, объект-имитация) – 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). Функция, выдающая константу, или случайную величину из допустимого диапазона значений.

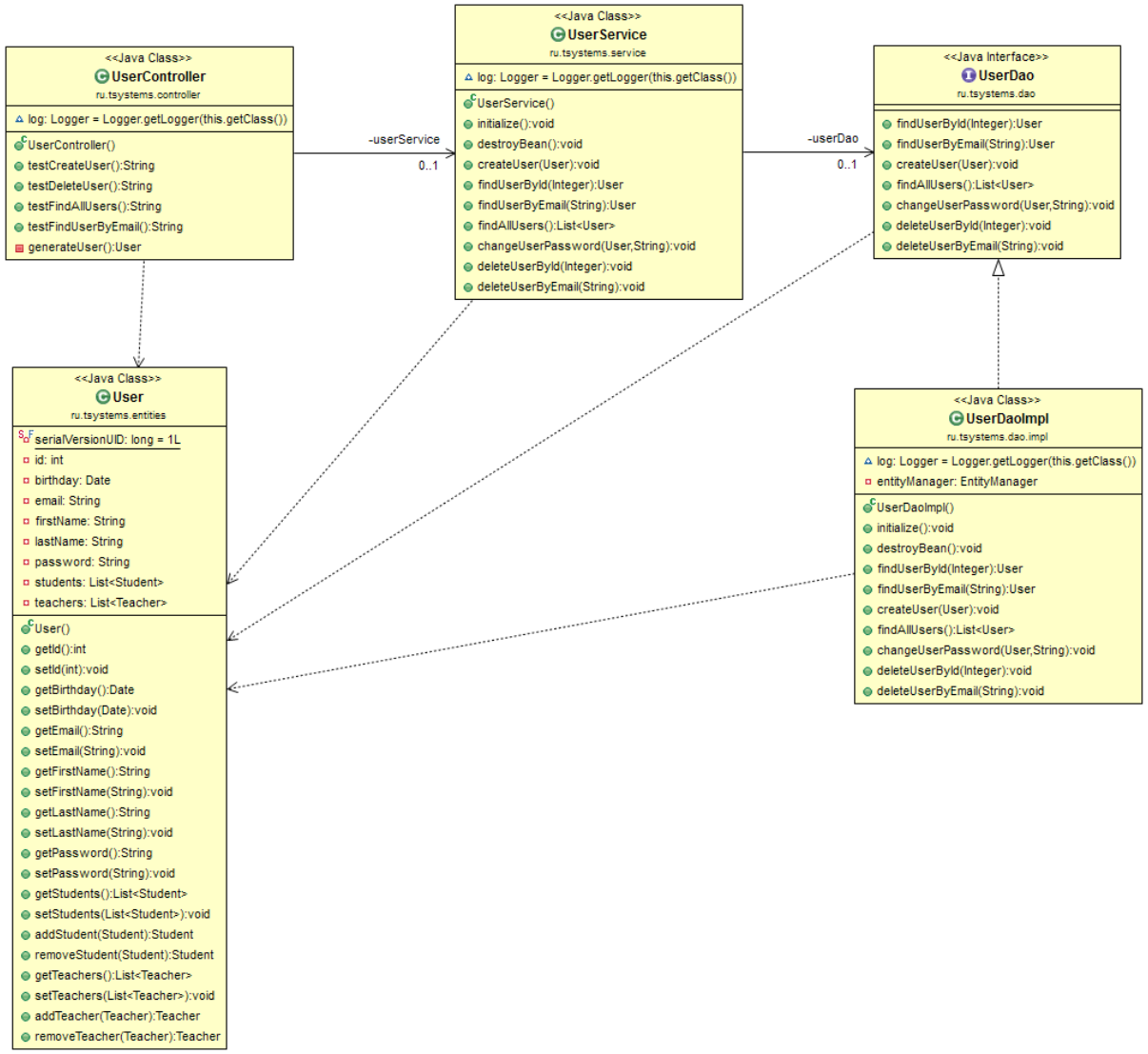
- Mock-объекты активно **используются в разработке через тестирование** (TDD).

Mock Object (2)

- **Моки** это такие **классы-заглушки** (и соответственно объекты), **которые позволяют избавиться от внешних зависимостей при модульном** (unit) **тестировании – ибо тестирование с зависимостями уже интеграционное или системное** и требует больших ресурсов, состояния данных и как следствие - большей сложности.

- С моками можно тестировать контроллеры, которые вызывают тяжёлые модели, у которых вся тяжёлая логика веб-сервисов, баз данных, парсеров и т.д., **но** которые уже покрыты тестами.

Mock Usage Example



Mockito (1)

Что умеет Mockito?

- Создавать моки.
- Определять значение, возвращаемое методом мока.
- Выбрасывать исключение при вызове метода мока.
- Проверять:
  - порядок вызовов.
  - количество вызовов.
  - отсутствие вызовов.

Mockito (2)

- **Задание результата.**
  - Используйте метод **when()** совместно со следующими методами:
    - **thenAnswer()**
    - **thenReturn()**
    - **thenThrow()**
  - Если возвращаемый объект не задать, то по умолчанию будут возвращаться null, 0, false.
- **Матчеры**.
  - Если нужно одинаковое выполнение для некоторого набора параметров, то используйте матчеры:
    - **when(obj.c(anyString(), anyString())).thenReturn(true);**

T · · Systems

Mockito (3)

- **Проверка вызова.**
  - Используйте метод **verify()**:
    - **verify(obj).c("", "");**
    - verify(obj, times(1)).c("", "");
    - verify(obj, atLeast(2)).c("", "");
    - verify(obj, atMost(2)).c("", "");
    - verify(obj, never()).c("", "");

PowerMock (1)

Расширение Mockito и EasyMock

- Позволяет создавать моки на:
  - static-методы
  - private-методы
  - final-методы
  - конструкторы

- **Назначение.**
  - Тестирование сторонних библиотек, к которым нет доступа на уровне исходников.
  - Быстрое тестирование собственного плохого кода без проведения длительного рефакторинга.

T · · Systems

PowerMock (2)

- **Использование.**
  - Добавляем перед тестом аннотацию:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({ ClassToBeMocked.class })
```

  - Либо используем JUnitRule (Junit 4.7+):

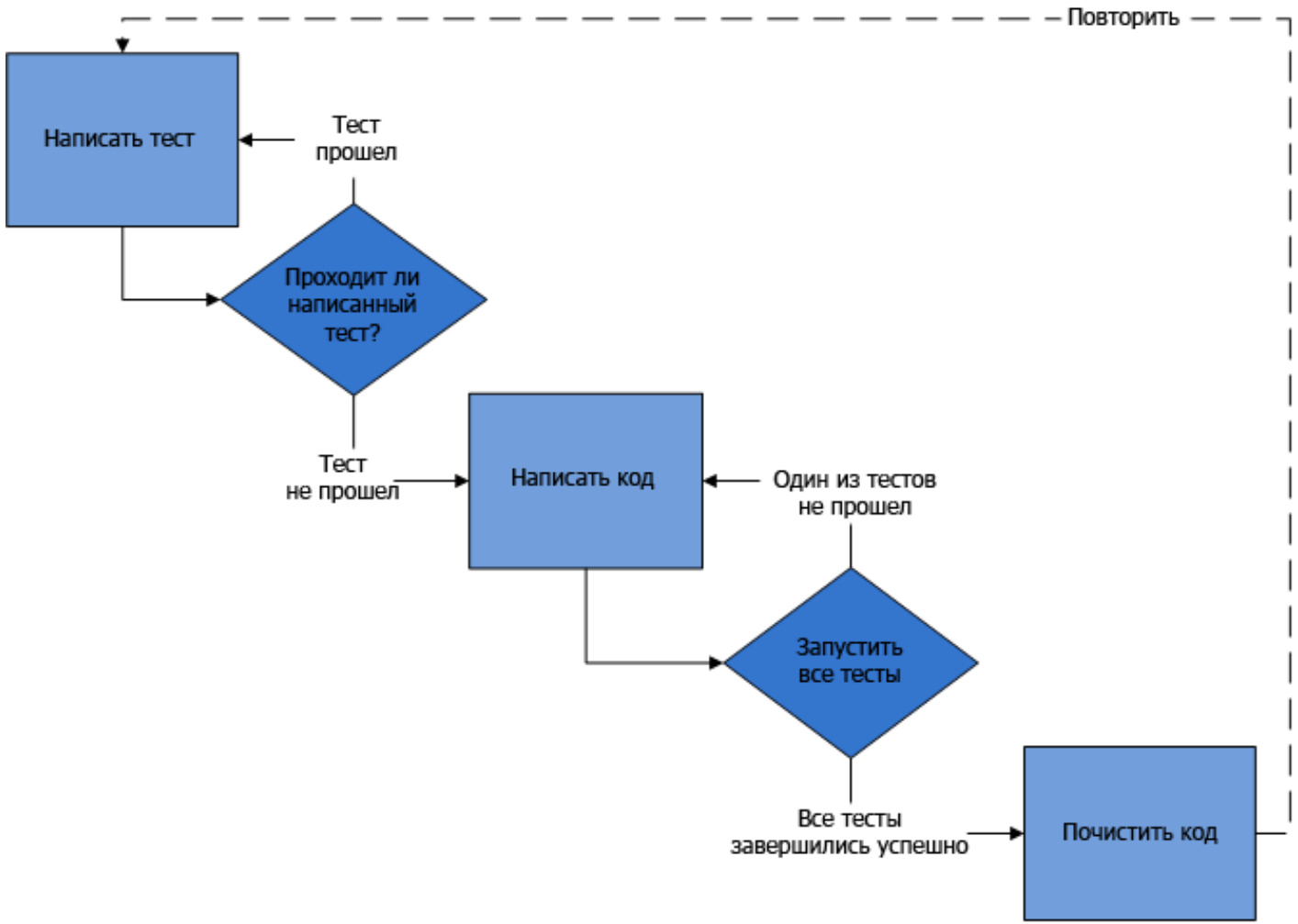
```
@PrepareForTest(X.class);
public class MyTest {
    @Rule
    PowerMockRule rule = new PowerMockRule();

    // Tests go here
}
```

Разработка через тестирование (1)

- **Разработка через тестирование** (test-driven development, TDD) – **техника** разработки программного обеспечения, которая **основывается на повторении очень коротких циклов разработки**: сначала пишется **тест**, покрывающий желаемое изменение, затем пишется **код**, который позволит пройти тест, и под конец проводится **рефакторинг** нового кода к соответствующим стандартам.

  - **Добавление теста** (фокусирует внимание девелопера на бизнес требованиях, а не на написании кода)
  - **Запуск тестов и получение отрицательных результатов**
  - **Написание кода под имеющиеся тесты**
  - **Запуск тестов и проверка их прохождения**
  - **Рефакторинг кода**

Разработка через тестирование (2)



T · · Systems

Спасибо за внимание!

T Systems