**МИНОБРНАУКИ РОССИИ**
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра программной инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О.Г. Локтионова
«24» _____ 2017 г.

# АВТОМАТИЗАЦИЯ ПРОЦЕССА ТЕСТИРОВАНИЯ ПРИ ОБЪЕКТНО – ОРИЕНТИРОВАННОМ ПРОЕКТИРОВАНИИ

Методические указания по выполнению лабораторной работы по дисциплине "Проектирование и архитектура программных систем" для студентов направления подготовки 09.03.04 "Программная инженерия"

Курск 2017

УДК 004.65

Составители: В.Г. Белов, Т.М. Белова

Рецензент
Кандидат технических наук, доцент кафедры программной инженерии
ЮЗГУ И.Н. Ефремова

**Автоматизация процесса тестирования при объектно – ориентированном проектировании**: методические указания по выполнению лабораторной работыпо дисциплине "Проектирование и архитектура программных систем" для студентов направления подготовки 09.03.04 "Программная инженерия" / Юго-Зап. гос. ун-т; сост.: В.Г. Белов, Т.М. Белова, – Курск, 2017. – 25 с.: ил. 21, табл. 1.

Изложена последовательность действий с программным обеспечениемJMock,EaseMock иRMock для тестирования программного обеспечения при объектно – ориентированном проектировании.

Материал предназначен для студентов направления подготовки бакалавров 09.03.04 «Программная инженерия», а также будет полезен студентам всех направлений подготовки, изучающим технологии тестирования пользовательского интерфейса веб-приложений.

Текст печатается в авторской редакции.

Подписано в печать 25.12.17. Формат 60x84 1/16.
Усл. печ. л. 1,4. Уч.-изд. л. 1,6. Тираж 100 экз. Заказ 4372. Бесплатно.
Юго-Западный государственный университет
305040, Курск, ул.50 лет Октября, 94.

Содержание

## 1 Объектно-ориентированное тестирование в Eclipse

При объектно-ориентированном тестировании используются фиктивные объекты имитируют поведение классов, написанных с единственной целью - управление выполнением кода во время тестирования.

## 2 Настройка jMock и RMock в Eclipse IDE

Сначала запустите интегрированную среду разработки Eclipse (integrated development environment - IDE). Затем создайте базовый Java™-проект, в который выполните импорт JAR-библиотек JUnit, jMock и RMock. Назовите Java-проект TestingExample. В перспективе Java выберите **Project>Properties**, а затем перейдите в закладку **Libraries**, как показано ниже на рисунке 1.

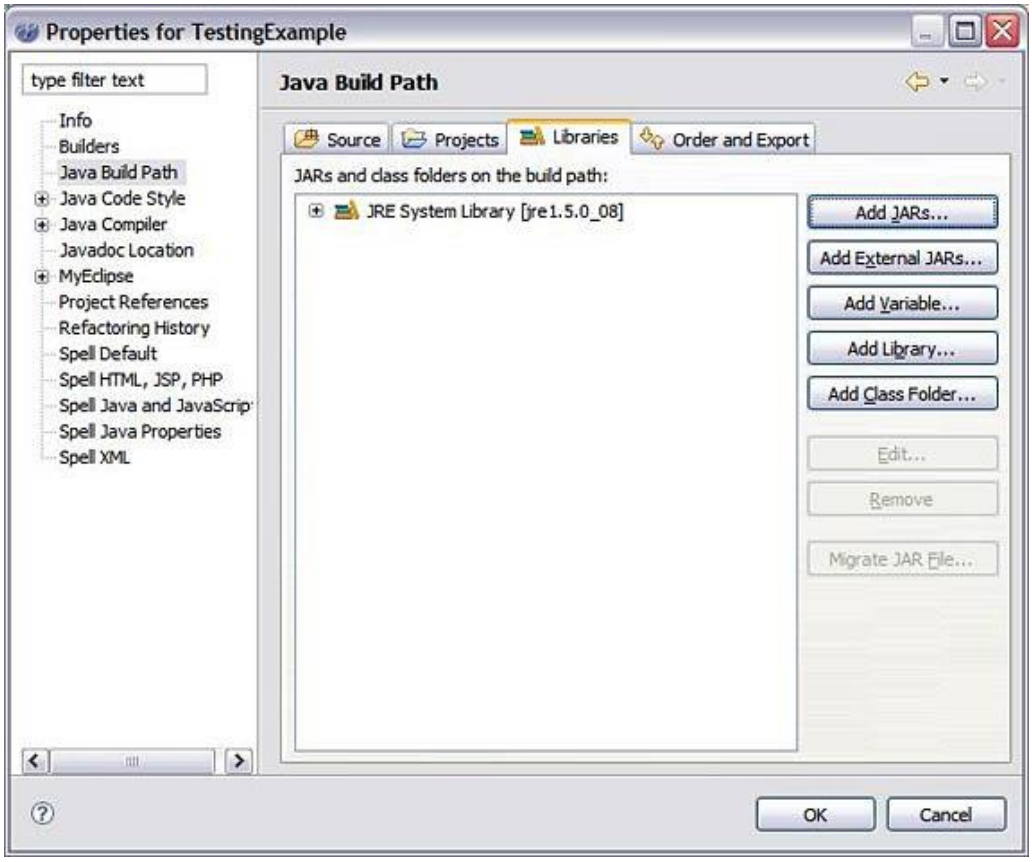


Рисунок 1- Редактирование свойств проекта TestingExample в Eclipse

Используйте кнопку **Add JARs**, если JAR-файлы указаны в Java class path (Java Runtime Environment (JRE), настроенная в Eclipse). Кнопка **Add Variable** работает с конкретным каталогом файловой системы (локальной или удаленной), где размещены ресурсы (включая JAR-файлы), на которые можно сослаться. Используйте кнопку **Add Library**, если нужно сослаться на такие специализированные ресурсы, которые используются в Eclipse по умолчанию или настроены на специализированную среду рабочей области Eclipse. Нажмите кнопку **Add Class Folder**, чтобы добавить ресурс из одной из папок существующих проектов, уже настроенных как часть проекта.

Для данного примера нажмите Add External JARs и найдите JAR-файлы jMock и RMock, которые уже загрузили. Добавьте их в проект. Нажмите кнопку OK, когда появится окно свойств, изображенное на рисунке 2.

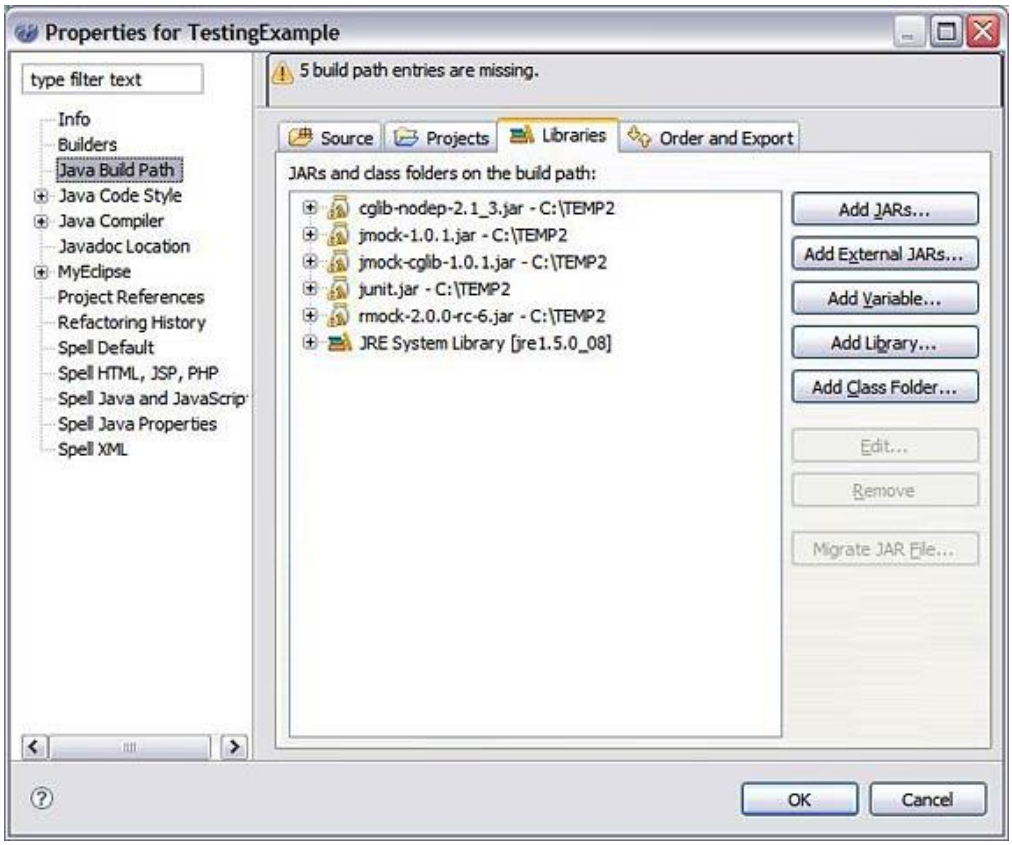


Рисунок 2 - JAR-файлы jMock и RMock, добавленные в проект TestingExample

## 3 Исходный код TestExample

В TestExample Project следует работать с исходным кодом четырех классов:

- ServiceClass.java,
- Collaborator.java,
- ICollaborator.java,
- ServiceClassTest.java.

Тестируемым классом является ServiceClass, который содержит один метод: runService(). Метод service принимает объект Collaborator, реализующий простой интерфейс ICollaborator. Один метод реализован в конкретном классе Collaborator: executeJob(). Collaborator - это класс, который вы должны имитировать соответствующим образом.

Четвертый класс - это тестовый класс ServiceClassTest (реализация максимально упрощена). На рисунке 3 показан исходный код этого четвертого класса.

```
public class ServiceClass {
    public ServiceClass(){
    //конструктор без аргументов
    }

    public Boolean runService(ICollaborator collaborator){
    if("success".equals(collaborator.executeJob())){
        return true;
    }
    else
    {
        return false;
    }
}
}
```

Рисунок 3 - Код примера класса Service

В классе ServiceClass блок кода if...else является простым логическим переходом, помогающим отобразить, почему тест завершится неудачно или успешно при выборе одного (а не другого) пути, в соответствии с ожидаемыми результатами. Исходный код класса Collaborator показан на рисунке 4.

```
public class Collaborator implements ICollaborator{
  publicCollaborator(){
     //конструктор без аргументов
  }
  public String executeJob(){
     return "success";
  }
}
```

Рисунок4 - Код примера класса Collaborator

Класс Collaborator с конструктором без аргументов и простой переменной String, возвращаемой из метода executeJob(), тоже не сложен. На рисунке 5 показан код класса ICollaborator.

```
public interface ICollaborator {
   public abstract String executeJob();
}
```

Рисунке 5 - Код класса ICollaborator

Интерфейс ICollaborator имеет один метод, который должен быть реализован в классе Collaborator.

Имея приведенный выше код, давайте перейдем к рассмотрению того, как можно успешно выполнить ваш тест класса ServiceClass в различных сценариях.

### 3.1 Сценарий 1: Использование jMock для имитации интерфейсов.

Тестирование метод service в классе ServiceClass осуществляется следующим образом. Предположим, что предметом тестирования является утверждение, что метод runService() не выполнялся, или, другими словами, что возвращенный Boolean-ре-

зультат равен false. В этом случае имитируется передаваемый в метод runService() объект ICollaborator для ожидания вызова его метода executeJob() и возврата строки, отличной от "success". Таким образом, вы гарантируете, что Boolean-строка false возвращается в тест.

Код класса ServiceClassTest, приведенный на рисунке 6, содержит логику теста.

```
import org.jmock.Mock;
import org.jmock.cglib.MockObjectTestCase;
public class ServiceClassTest extends MockObjectTestCase {
   private ServiceClassserviceClass;
   private Mock mockCollaborator;
   private ICollaborator collaborator;

   public void setUp(){
      serviceClass = new ServiceClass();
      mockCollaborator = new Mock(ICollaborator.class);
   }

   public void testRunServiceAndReturnFalse(){
      mockCollaborator.expects(once()).method\
          ("executeJob").will(returnValue("failure"));
      collaborator = (ICollaborator)mockCollaborator.proxy();
      boolean result = serviceClass.runService(collaborator);
      assertFalse(result);
   }
}
```

Рисунок 6 -код примера класса ServiceClassTest для сценария 1

### 3.1.1 Когда нужно писать собственные тесты

Лучшим способом выполнения своих собственных экспериментов с любой интегрированной средой имитационного тестирования является динамичный подход test-first. Сначала создайте тест и установите ожидаемые результаты. Только после неудачного выполнения теста необходимо написать реализацию для корректировки теста. Если тест работает, пишется другой тест для проверки функциональности, добавляемый в тестируемый класс позже.

Обычно хорошей идеей является включение в тесты метода setUp(), если в различных примерах тестов выполняются общие операции. Метод tearDown() тоже годится, но он не столь необходим до тех пор, не будуе выполняться интегрированные тесты.

Также следует обратить внимание на то, что в jMock и RMock среда проверяет все ожидаемые результаты по всем фиктивным объектам в конце (или во время) выполнения теста. Нет реальной необходимости включать метод verify() для каждого ожидаемого результата. При выполнении теста как JUnit, тест завершается успешно, как показано на рисунке 7.

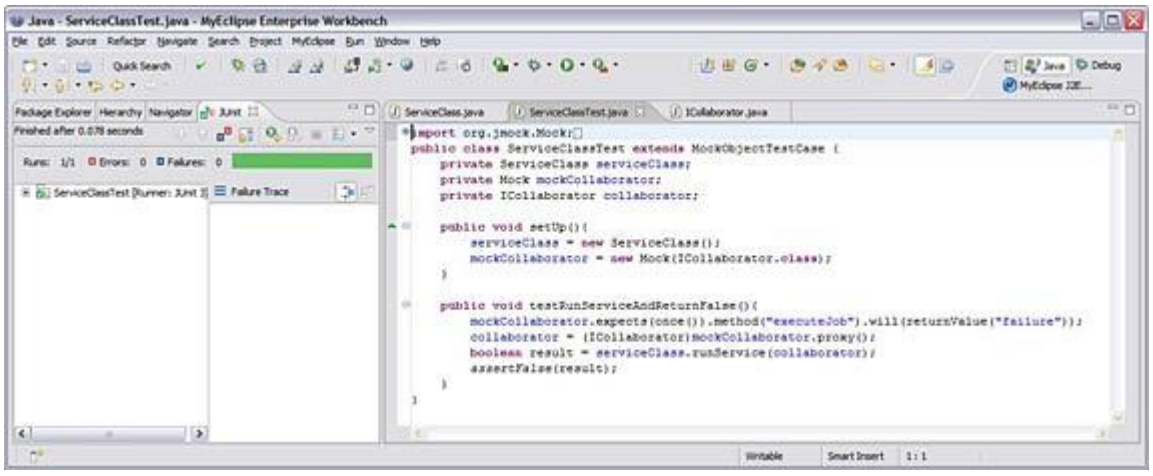


Рисунок 7 - Успешное выполнение теста сценария 1

Класс ServiceTestClass расшряет класс org.jmock.cglib.MockObjectTestCase jMockCGLIB. Класс mockCollaborator - это простой класс org.jmock.JMock. Обычно есть два способа создания фиктивных объектов в jMock:

- Для имитации интерфейса используется новый метод Mock(Class.class).
- Для имитации конкретного класса используется метод mock(Class.class, "identifier").

Важно отметить, как имитируемый proxy передается в метод runService() класса ServiceClass. В jMock можно извлечь реализации proxy из созданных фиктивных объектов, для которых ожидаемые результаты уже были установлены. Это будет важно в следующих сценариях данной статьи, особенно при работе с RMock.

### 3.2 Сценарий 2: Использование jMock для имитации конкретного класса с конструктором по умолчанию

Предположим, что метод runService() в классе ServiceClass принимает только конкретные реализации класса Collaborator. Будет ли достаточно jMock для проверки того, что предыдущий тест выполнился успешно без изменения ожидаемых результатов? Да, поскольку можно создать класс Collaborator просто по умолчанию.

Измените метод runService() в классе ServiceClass, как показано на рисунке 8.

```
publicclassServiceClass {
   publicServiceClass(){
   //конструкторбезаргументов
   }
 public booleanrunService(Collaborator collaborator){
   if("success".equals(collaborator.executeJob())){
      return true;
   }
   else{       return false;
   }}}
```

Рисунок 8 - Измененный класс ServiceClass для сценария 2

Логическое ветвление if...else класса ServiceClass остается без изменений (для ясности). Конструктор без аргументов также все еще на месте. Обратите внимание на то, что не всегда есть необходимость в таких логических конструкциях, как циклы while...do или for, для соответствующего тестирования методов класса. Поскольку имеются исполнения методов объектов, используемых классом, достаточно простых ожидаемых результатов имитации для тестирования этих исполнений.

Необходимо также изменить класс ServiceClassTest для данного сценария, как показано на рисунке 9.

```
        ...
        private ServiceClassserviceClass;
            private Mock mockCollaborator;
            private Collaborator collaborator;

            public void setUp(){
                serviceClass = new ServiceClass();
                mockCollaborator = mock
                (Collaborator.class, "mockCollaborator");
            }

            public void testRunServiceAndReturnFalse(){
                mockCollaborator.expects(once()).
                method("executeJob").
                will(returnValue("failure"));
                collaborator =
                (Collaborator)mock
                Collaborator.proxy();
                boolean result =
                serviceClass.runService(collaborator);
                assertFalse(result);
            }
}
```

Рисунок 9 - Измененный класс ServiceClassTest для сценария 2

Здесь следует отметить несколько моментов. Во-первых, сигнатура метода runService() изменилась. Вместо приема интерфейса ICollaborator он принимает теперь реализацию конкретного класса (класса Collaborator). Это важно для работы тестовой интегрированной среды (пример по своей природе является анти-полиморфным, и мы передаем конкретный класс только в этом примере; так нельзя делать при объектно-ориентированном подходе).

Во-вторых, изменился способ имитации класса Collaborator. CGLIB-библиотека jMock предоставляет возможность имитировать конкретный класс. Дополнительный String-параметр для метода mock() jMock CGLIB используется в качестве идентификатора создаваемого фиктивного объекта. При использовании jMock (и, конечно же, RMock) уникальные идентификаторы необходимы для каждого имитируемого объекта в одном контрольном примере. Это верно для фиктивных объектов, определенных в общем методе setUp() или в реальном методе test.

В-третьих, оригинальные ожидаемые результаты метода test не изменились. Ложное утверждение все еще необходимо для прохождения теста. Это важно, поскольку демонстрирует, что используемые интегрированные среды тестирования достаточно гибки для того, чтобы подстроиться под различные входные параметры, одновременно обеспечивая постоянные результаты тестирования. Их истинные ограничения проявляются, когда входные параметры нельзя подстроить для формирования таких же результатов.

Теперь, перезапустите тест как JUnit-тест. Тест завершается успешно, как показано на рисунке 10.

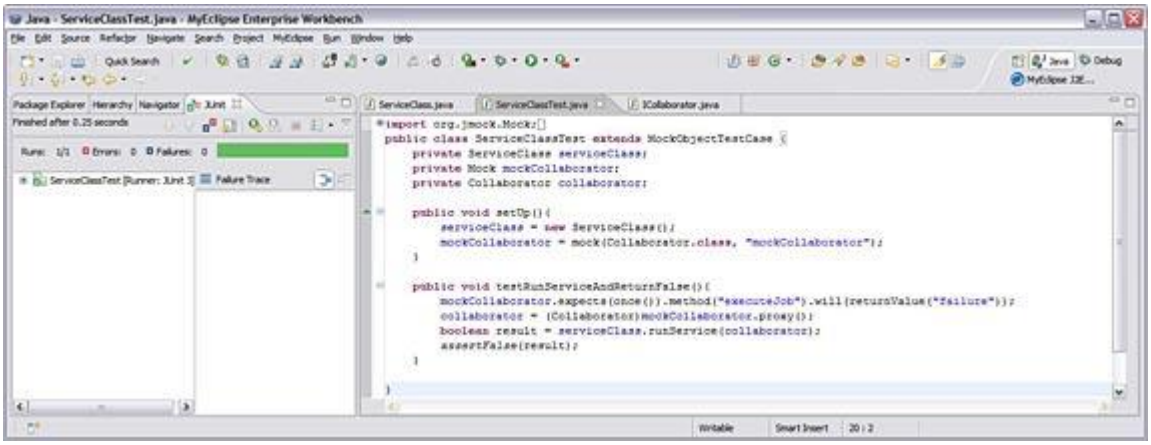


Рисунок 10 - Выполнение теста сценария 2

Следующий сценарий немного усложняется. Чтобы облегчить ситуацию, которая может показаться тяжелой, используется интегрированная среда RMock.

### 3.3 Сценарий 3: Использование jMock и RMock для имитации конкретного класса с конструктором не по умолчанию.

Начните, как и прежде, с попытки использовать jMock для имитации объекта Collaborator, только на этот раз Collaborator не имеет конструктора без аргументов по умолчанию. Обратите внимание на то, что поддерживается ожидаемый результат теста Boolean false.

Также предположим, что объект Collaborator требует строку и примитивный тип int в качестве параметров, передаваемых в конструктор. В листинге на рисунке 11 показаны изменения в объекте Collaborator.

```
public class Collaborator{
  private String collaboratorString;
  private intcollaboratorInt;

  public Collaborator(String string, int number){
    collaboratorString = string;
    collaboratorInt = number;
  }
  public String executeJob(){
   return "success";
  }
}
```

Рисунок 11 - Измененный класс Collaborator для сценария 3

Конструктор класса Collaborator все еще довольно прост. Поля класса устанавливаются равными входным параметрам. Другой логики здесь не нужно, и функция executeJob() остается такой же.

Перезапустите тест с неизменными остальными компонентами примера. Результат - фатальная ошибка выполнения теста, как показано на рисунке 12.

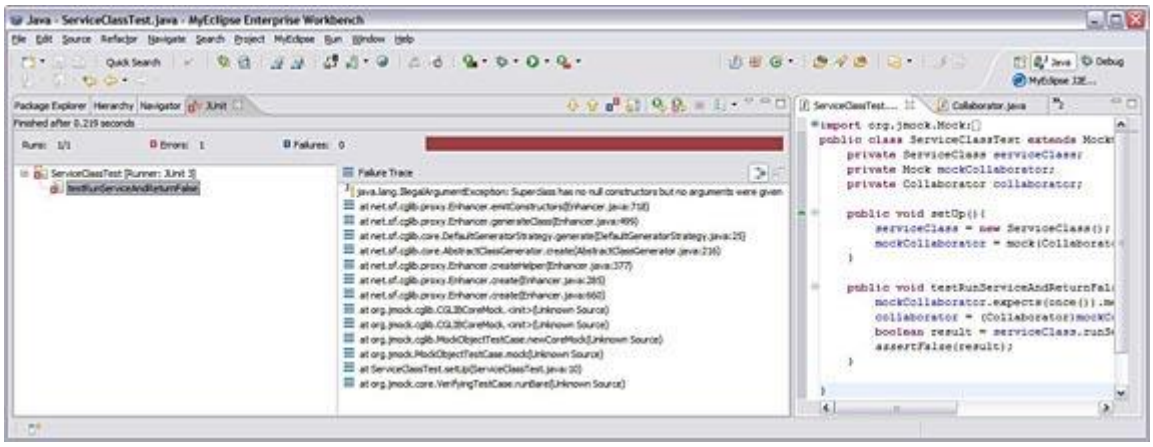

Рисунок 12 - Ошибка теста сценария 3

Приведенный выше тест был выполнен как простой JUnit-тест без покрытия кода (code coverage). Вы можете выполнить любые тесты, приведенные в данной статье, в инструментальных системах с покрытием кода (например, Cobertura или EclEmma). Однако существуют некоторые проблемы при выполнении RMock-тестов с покрытием кода внутри Eclipse (см. таблицу 1). На рисунке 13 показан фрагмент кода, демонстрирующий реальную трассировку стека.

```
...Superclass has no null constructors but no arguments were given
   at net.sf.cglib.proxy.Enhancer.emitConstructors(Enhancer.java:718)
   at net.sf.cglib.proxy.Enhancer.generateClass(Enhancer.java:499)
   atnet.sf.cglib.core.DefaultGeneratorStrategy.
   generate(DefaultGeneratorStrategy.java:25)
   atnet.sf.cglib.core.AbstractClassGenerator.
   create(AbstractClassGenerator.java:216)
   at net.sf.cglib.proxy.Enhancer.create
   Helper(Enhancer.java:377)
   at net.sf.cglib.proxy.Enhancer.create
   (Enhancer.java:285)
   at net.sf.cglib.proxy.Enhancer.create
   (Enhancer.java:660)
```

Рисунок 13 - Трассировка стека для неудачного теста в сценарии 3

Причина неудачного выполнения заключается в том, что jMock не может создать жизнеспособный фиктивный объект из определения класса, в котором нет конструктора без аргументов. Единственным способом создания экземпляра объекта Collaborator является предоставление двух аргументов. Вам теперь придется найти способ предоставления аргументов в процесс создания экземпляра фиктивного объекта для достижения аналогичного эффекта. Именно поэтому используется RMock.

### 3.3.1 Корректировка неудачного теста с использованием интегрированной среды тестирования RMock

Для корректировки теста нужно сделать несколько изменений. Они могут показаться значительными, но, по существу, это относительно простая работа для использования возможностей обеих интегрированных сред в целях интеграционного тестирования.

Первым необходимым изменением является создание тестового класса как RMock Test Case, а не jMock CGLIB Test Case. Цель - возможность более легкой настройки фиктивных объектов, принадлежащих RMock, в самих тестах, а также (что более важно) во время их начальной настройки. Опыт показывает, что легче создать и использовать фиктивные объекты из обеих интегрированных сред, когда весь объект Test Case, из которого расширяется тестовый класс, принадлежит RMock. Более того,

на первый взгляд несколько легче быстро определить поток (flow) фиктивных объектов (*поток* здесь применяется для описания ситуации, в которой фиктивный объект используется в качестве параметра или даже как возвращаемый тип из других фиктивных объектов).

Второе необходимое изменение - создать (как минимум) массив объектов, содержащий реальные значения параметров, передаваемых в конструктор класса Collaborator. Также возможно (для ясности) включить class-types массив типов, принимаемых конструктором, и передать этот массив, так же как и только что описанный массив объектов, в качестве параметров для создания экземпляра фиктивного объекта Collaborator.

Третье изменение затрагивает создание одного или нескольких ожидаемых результатов в фиктивном объекте RMock с корректным синтаксисом. И четвертым, последним необходимым изменением является перевод фиктивного объекта RMock из состояния record в состояние ready.

### 3.3.2 Реализация изменений RMock

На рисунке 14 показаны окончательные изменения в классе ServiceClassTest. Также показано использование RMock и ее функциональности.

```
...
importcom.agical.rmock.extension.junit.
RMockTestCase;
public class ServiceClassTest extends
RMockTestCase {
    private ServiceClassserviceClass;
   private Collaborator collaborator;

   public void setUp(){
      serviceClass = new ServiceClass();
      Object[] objectArray = new
      Object[]{"exampleString", 5};
            collaborator =(Collaborator)intercept
            (Collaborator.class, objectArray,
            "mockCollaborator");
   }
       public void testRunServiceAndReturnFalse(){
      collaborator.executeJob();
```

```
        modify().returnValue("failure");
        startVerification();
        boolean result = serviceClass.
        runService(collaborator);
        assertFalse(result);
    }}
```

Рисунок 14 - Корректировка класса ServiceClassTest для сценария 3

Прежде всего, обратите внимание на то, что ожидаемые результаты теста не изменились. Импорт класса RMockTestCase извещает о появлении функциональности интегрированной среды RMock. Далее тестовый класс расширяет RMock Test Case, а не Mock Object Test Case.

### 3.3.3 Альтернатива использованию метода intercept()

Используя RMock, можно применить метод intercept() для имитации только конкретных классов. Метод RMock mock() можно применять для имитации конкретных классов и интерфейсов. Используйте interface(), когда нужно имитировать не много методов - только те, которые действительно имеют значение. Считайте этот метод улучшенным методом mock().

В методе setUp() создается экземпляр массива объектов с реальными значениями, которые нужны конструктору класса Collaborator. Этот массив передается полностью в метод intercept() RMock, для того чтобы помочь создать экземпляр фиктивного объекта. Сигнатура метода аналогична сигнатуре метода jMock CGLIB mock(), поскольку оба метода принимают в качестве аргументов уникальные идентификаторы фиктивных объектов. Необходимо приведение типов класса фиктивного объекта в тип Collaborator, поскольку метод intercept() возвращает тип Object.

Внутри самого тестового метода testRunServiceAndReturnFalse()можно увидеть немного больше изменений. Вызывается метод executeJob() фиктивного объекта Collaborator. На этом этапе фиктивный метод находится в состоянии record, то есть, вы просто определяете вызовы методов, которые он будет ожидать при выполнении. Соответственно фиктивный объект записывает ожидаемые результаты. Следующая строка

- это уведомление фиктивному объекту при появлении метода executeJob() возвратить строковое значение **failure**. Следовательно, используя RMock, вы устанавливаете ожидаемый результат простым вызовом метода вне фиктивного объекта (и передавая все параметры, которые могут понадобиться), затем изменяете этот ожидаемый результат для подстройки всех возвращаемых типов соответствующим образом.

Наконец, вызывается метод startVerification() RMock для перевода фиктивного объекта Collaborator в состояние ready. Фиктивный объект теперь готов для использования в классе ServiceClass в качестве реального объекта. Метод абсолютно необходим и должен вызываться, для того чтобы избежать ошибок инициализации теста.

### 3.3.4 Тестирование изменений

Опять перезапустите ServiceClassTest для получения окончательного положительного результата: предоставленные вами во время создания экземпляра фиктивного объекта параметры сделали все необходимое. На рисунке 15 показан положительный зеленый цвет JUnit.

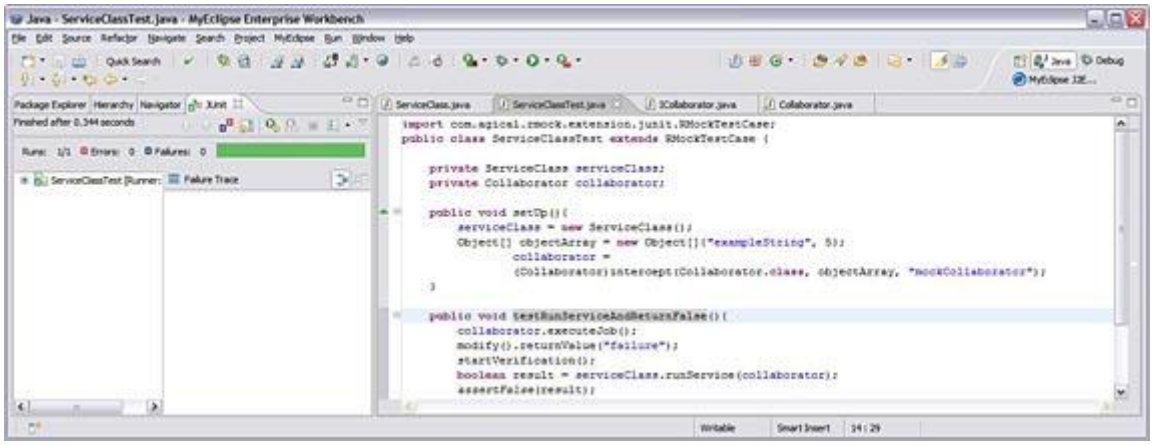

Рисунок 15 - Успешное завершение теста в сценарии 3 с использованием RMock

Строка кода assertFalse(result) представляет такой же ожидаемый результат, что и в сценарии 1, а RMock обеспечивает успех выполнения теста, как это ранее делала среда jMock. По многим причинам это важно, но более важным моментом здесь является то,

что *динамичный* (agile) принцип корректировки неудачного теста применяется без изменения ожидаемых результатов теста. Единственным различием является использование альтернативной интегрированной среды.

В следующем сценарии вы будете использовать обе среды jMock и RMock в особой ситуации. Ни одна среда сама по себе не обеспечила бы корректного результата. Нужно будет организовать в тесте определенного рода союз обеих сред.

### 3.4 Сценарий 4: Совместная работа jMock и RMock

Две интегрированные среды должны работать совместно для достижения определенного результата. В противном случае правильно сформированный тест будет постоянно завершаться неудачно. Существует несколько случаев, в которых не имеет значения, какую среду использовать (jMock или RMock), например, когда интерфейс или класс, который вы хотите имитировать, существует в подписанномJAR-архиве. Это редкая ситуация, но она может возникнуть при тестировании кода, написанного с использованием API для защищенных лицензированных продуктов (обычно готовое программное обеспечение какого-либо рода).

На рисунках 16-17 показан пример, в котором обе интегрированные среды работают в одном контрольном примере.

```
public class MyNewClassTest extends RMockTestCase{
 private MyNewClassmyClass;
private MockObjectTestCasetestCase;

3.5     private Collaborator collaborator;

private Mock mockClassB;
   public void setUp(){
      myClass = new MyNewClass();

      testCase = new MyMockObjectTestCase();

      mockClassB = testCase.mock
      (ClassB.class, "mockClassB");
```

```
            mockClassB.expects
```

Рисунок 16 - Пример теста для сценария 4

```
            (testCase.once()).method("wierdMethod").will(testCase.return-
Value("passed"));
        Class[] someClassArray =
            new Class[]{String.class, ClassA.class, ClassB.class};
            Object[] someObjectArray = new Object[]
               {"someArbitraryString",
               new ClassA(), (ClassB)mockClassB.proxy()};
       collaborator = (Collaborator)intercept(Collaborator.class, someClassArray,
                  someObjectArray, "mockCollaborator");
         }

         public void testRMockAndJMockInCollaboration(){
            startVerification();
            assertTrue(myClass.executeJob(collaborator));
         }

         private class MyMockObjectTestCase
          extends MockObjectTestCase{}

         private class MyNewClass{
            public booleanexecuteJob(Collaborator collaborator){
               collaborator.executeSomeImportantFunction();
               return true;
            }
         }
}
```

Рисунок 17 - Пример теста для сценария 4

В методе setUp() создается экземпляр нового "testcase" на основе privateinner класса, созданного для расширения объекта jMock-CGLIB MockObjectTestCase. Эта небольшая дополнительная работа необходима для хранения всего тестового класса как объекта RMock TestCase, одновременно обладающего всей функциональностью jMock. Например, вы установите ожидаемые результаты jMock как testCase.once(), а не как once(), поскольку объект TestClassрасширяет RMockTestCase.

Создается фиктивный объект, основанный на классе ClassB и предоставляющий ожидаемый результат. Затем вы используете его для помощи в создании экземпляра фиктивного объекта RMock Collaborator. Тестируемым классом является MyNewClass (показанный здесь как privateinner класс). Опять же, его метод executeJob() принимает объект Collaborator и выполняет метод executeSomeImportantFunction().

На рисунках 18 и 19 показан код ClassA и ClassB соответственно. ClassA - это простой класс без реализации, в то время как ClassB демонстрирует минимум деталей, иллюстрирующих ситуацию.

```
public class ClassA{}
```

Рисунок 18 –Класс ClassA

Этот класс является просто фиктивным (dummy) классом, который я использую, чтобы подчеркнуть необходимость RMock для имитирующих классов, конструкторы которых принимают объектные параметры.

```
public class ClassB{
        public ClassB(){}
        public String wierdMethod(){
            return "failed";
        }
}
```

Рисунок 19 –Класс ClassB

Метод wierdMethod класса ClassB возвращает failed. Это важно, поскольку класс должен возвращать другую строку для успешного прохождения теста.

На рисунке 20 показан наиболее важный фрагмент примера теста - класс Collaborator.

```
        public class Collaborator {
           private String  _string;
           private ClassA _classA;
           private ClassB _classB;

           public Collaborator(String string, ClassAclassA,
           ClassBclassB) throws Exception{
               _string = string;
              _classA = classA;
              if(classB.wierdMethod().equals("passed")){
                 _classB =classB;
              }
              else{throw new Exception
                    ("Something bad happened");
              }
           }

           public void executeSomeImportantFunction(){
           }
}
```

Рисунок 20 - Класс Collaborator

Во-первых, обратите внимание на то, что был имитирован класс ClassB, используя интегрированную среду jMock. В RMock не существует реального способа извлечь и использовать proxy из фиктивного объекта для применения где-нибудь в тестовом методе setUp(). В RMockproxy-объект появляется только *после* вызова метода startVerification(). Преимущество здесь на стороне jMock, поскольку вы *можете* получить все, что надо, для настройки других фиктивных объектов, когда они должны возвратить объекты, которые сами являются фиктивными.

Во-вторых, обратите внимание на то, что, с другой стороны, нельзя было использовать интегрированную среду jMock для имитации класса Collaborator. Причина состоит в том, что этот класс не имеет конструктора без аргументов. Более того, в его конструкторе есть определенная логика, которая определяет, может ли, прежде всего, быть

получен экземпляр класса. Фактически метод wierdMethod() в ClassB должен возвращать **passed** для создания экземпляра объекта Collaborator. Однако обратите внимание на то, что *по умолчанию* метод всегда возвращает failed. Существует очевидная необходимость имитировать ClassB для успешного выполнения теста.

Также, в отличие от предыдущих примеров, массив классов в данном сценарии включен как дополнительный параметр метода intercept(). Это не является острой необходимостью, но он служит ключом для быстрой идентификации используемых объектных классов при создании экземпляра тестового объекта RMock.

Выполните новый контрольный пример. На этот раз результаты будут успешными. Рисунок 21 демонстрирует удачное завершение.

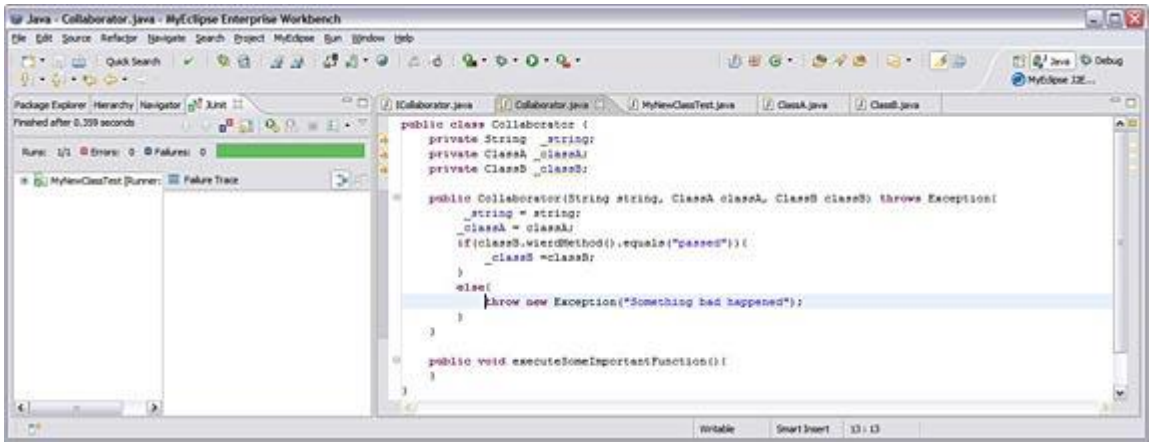


Рисунок 21 - Успешное выполнение теста для сценария 4 с использованием совместной работы RMock и jMock

Фиктивный объект Collaborator настроен корректно и выполнение объекта mockClassB приносит ожидаемый результат.

## 4 Краткий обзор различий инструментальных средств тестирования

Как можно заметить, в рассмотренных сценариях обе среды (jMock и RMock) являются мощными инструментами для тестирования Java-кода. Однако всегда существуют ограничения в случае использования других инструментальных программ, ис-

пользуемых в процессе разработки и тестировании. Кроме того, доступны другие программы тестирования, но ни одна из них не работает так хорошо (в Java-технологии), как RMock и jMock.

В таблице 1 приведены некоторые отличия двух интегрированных сред и возможные проблемы, возникающие время от времени, в частности в среде Eclipse.

Таблица 1 - Различия между интегрированными средами тестирования RMock и jMock

| **Стиль имитации теста** | **jMock** | **RMock** |
|---|---|---|
| **Можно имитировать интерфейсы** | Да: Новый метод Mock() | Да: Метод mock() |
| **Можно имитировать конкретные классы** | Да: Метод mock() с CGLIB | Да: Меод mock() или intercept() |
| **Можно имитировать любой конкретный класс** | Нет: Должен присутствовать конструктор без аргументов | Да |
| **Можно получить прокси в любое время** | Да | Нет: Только после состояния ready startVerification() |

| **Стиль имитации теста** | **jMock** | **RMock** |
|---|---|---|
| **Проблемы с другими подключаемыми модулями Eclipse** | Нет: Проблем не обнаружено | Да: Конфликты оперативной памяти с подключаемым модулем CoverClipse для Eclipse |