Лекция 24

Тестирование. Пакет JUnit

Сегодня большую популярность приобретает test-driven development(TDD), техника разработки ПО, при которой сначала пишется тест на определенный функционал, а затем пишется реализация этого функционала.

В результате код не только написан и протестирован, но тесты как бы неявно задают требования к функционалу, а также показывают пример использования этого функционала.

Самым известным фреймворком является JUnit. Используется он в двух вариантах JUnit 3 и JUnit 4

JUnit 3

Для создания теста нужно унаследовать тест-класс от **TestCase**, переопределить методы **setUp** и **tearDown** если надо, а также создать тестовые методы(должны начинаться с test).

При запуске теста сначала создается экземпляр тест-класса (для каждого теста в классе отдельный экземпляр класса), затем выполняется метод setUp, запускается сам тест, ну и в завершение выполняется метод tearDown.

Если какой-либо из методов выбрасывает исключение, тест считается провалившимся.

Тестовые методы должны быть public void, могут быть static.

Сами тесты состоят из выполнения некоторого кода и проверок.

Проверки чаще всего выполняются с помощью класса Assert хотя иногда используют ключевое слово assert.

Рассмотрим пример.

Пусть имеется класс, который имеет два метода - нахождение факториала и суммы двух чисел:

```
public class MathFunc {
  private int variable;
  public MathFunc() { variable = 0; }
  public MathFunc(int var) { variable = var; }
```

```
public int getVariable() { return variable; }

public void setVariable (int variable){
   this.variable = variable;
}

public long factorial() {
  long result = 1;
  if (variable > 1) {
     for (int i=1; i<=variable; i++)
           result = result*i;
  }
  return result;
}

public long plus(int var) {
  long result = variable + var;
  return result;
 }
}
```

Напишем для этого класса тесты, используя JUnit 3.

Удобнее всего, писать тесты, рассматривая некий класс как черный ящик, писать отдельный тест на каждый значимый метод в этом классе, для каждого набора входных параметров какой-то ожидаемый результат.

Для написания тестового класса нужно создать наследника junit.framework.TestCase.

Затем необходимо определить конструктор, принимающий в качестве параметра строку (String) и передающую ее родительскому классу.

```
import junit.framework.*;
public class TestClass extends TestCase {
 public TestClass(String testName) { super(testName);
 }
 public void testFactorialNull() {
    MathFunc math = new MathFunc();
    assertTrue(math.factorial() == 1);
 }

 public void testFactorialPositive() {
  MathFunc math = new MathFunc(5);
  assertTrue(math.factorial() == 120);
 }

 public void testPlus() {
   MathFunc math = new MathFunc(45);
   assertTrue(math.plus(123) == 168);
 }
}
```

Метод assertTrue проверяет, является ли результат выражения верным.

Некоторые другие методы, которые могут пригодиться - **assertEquals, assertFalse, assertNull, assertNotNull, assertSame**.

Стоит заметить, что **assertEquals** и **assertSame** отличаются:

assertSame(Object expect, Object actual)

Эквивалентно **expect == actual**

assertEquals(Object expect, Object actual)

Эквивалентно **expect.equals(actual).**

Для того, чтобы объединить тесты, можно воспользоваться классом **TestSuite** с его методом addTest.

Наконец, для запуска всех тестов нужно воспользоваться **TestRunner**.

Можно использовать текстовый junit.textui.TestRunner

(есть также графические версии - junit.swingui.TestRunner, junit.awtui.TestRunner).

В итоге метод main имеет вид:

```
import junit.framework.*;
import junit.textui.*;

public class Main{
 public static void main(String[] args) {
   TestRunner runner = new TestRunner();
   TestSuite suite = new TestSuite();
   suite.addTest(new TestClass(“testFactorialNull”));
   suite.addTest( new TestClass(“testFactorialPositive”));
    suite.addTest(new TestClass(“testPlus”));
    runner.doRun(suite);
 }
}
```

На экране получим:

Time: 0

Ok(3 tests)

Для тестирования более комплексных классов, могут пригодится методы setUp и tearDown.

Первый метод может проинициализировать один или несколько экземпляров тестируемого класса для использования в нескольких TestCases, второй метод отпускает захваченные при инициализации ресурсы.

Наследуя тест-класс от **ExceptionTestCase**, можно проверить что-либо на выброс исключения.

JUnit 4

Здесь была добавлена поддержка новых возможностей из Java 5, тесты теперь могут быть объявлены с помощью аннотаций.

При этом существует обратная совместимость с предыдущей версией фреймворка.

Основные аннотации

- Аннотация **@Before** обозначает методы, которые будут вызваны до исполнения теста, методы должны быть **public void.**

  Здесь обычно размещаются предустановки для теста.

- Аннотация **@BeforeClass** обозначает методы, которые будут вызваны до создания экземпляра тест-класса, методы должны быть **public static void**.

  Имеет смысл размещать предустановки для теста в случае, когда класс содержит несколько тестов использующих различные предустановки, либо когда несколько тестов используют одни и те же данные.

- Аннотация **@After** обозначает методы, которые будут вызваны после выполнения теста, методы должны быть **public void**. Здесь размещаются операции освобождения ресурсов после теста.

- Аннотация **@AfterClass** связана по смыслу с **@BeforeClass**, но выполняет методы после теста, как и в случае с **@BeforeClass**, методы должны быть **public static void**.

- Аннотация **@Test** обозначает тестовые методы. Как и ранее, эти методы должны быть **public void**. Здесь размещаются сами проверки. Кроме того, у данной аннотации есть два параметра, **expected** — задает ожидаемое исключение и **timeout** — задает время, по истечению которого тест считается провалившимся.

Пример:

```
@Test(expected = NullPointerException.class)
public void testToHexStringWrong() {
    StringUtils.toHexString(null);
}

@Test(timeout = 1000)
public void infinity() { while (true); }
```

- Если какой-либо тест по какой-либо серьезной причине нужно отключить (например, этот тест постоянно валится, но его исправление отложено) его можно зааннотировать **@Ignore**.

Пример:

```
@Ignore
@Test(timeout = 1000)
public void infinity() { while (true); }
```

Рассмотрим пример:

```
@RunWith(JUnit4.class)
public class JUnit4Test extends Assert {
private final Map<String, byte[]> toHexStringData
                                = new HashMap<String, byte[]>()
@Before
public void setUpToHexStringData() {
  toHexStringData.put("", new byte[0]);
  toHexStringData.put("01020d112d7f",
                           new byte[] { 1, 2, 13, 17, 45, 127 });
  toHexStringData.put("00fff21180",
                              new byte[] { 0, -1, -14, 17, -128 });

//...
}
```

```
@After
public void tearDownToHexStringData() {
        toHexStringData.clear();
}
@Test
public void testToHexString() {
  for (Map.Entry<String, byte[]> entry :
                                          toHexStringData.entrySet()){
    final byte[] testData = entry.getValue();
    final String expected = entry.getKey();
    final String actual = StringUtils.toHexString(testData);
    assertEquals(expected, actual);
  }
}
}
```

Для того, чтобы запустить данный тест в командной строке необходимо ввести

```
>java -cp .;d:\distribs\java\junit-4.11.jar;
                    d:\distribs\java\hamcrest-all-1.3.jar
          org.junit.runner.JUnitCore   JUnit4Test
```

Запуск теста.

Запуск теста можно сконфигурировать с помощью **@RunWith.**

При этом класс, указанный в аннотации должен наследоваться от **Runner**.

Рассмотрим различные “запускалки”

JUnit4 — предназначен для запуска JUnit 4 тестов.

JUnit38ClassRunner предназначен для запуска тестов, написанных с использованием JUnit 3.

Categories — попытка организовать тесты в категории (группы).

Для этого тестам задается категория с помощью **@Category**, затем настраиваются запускаемые категории тестов.

Это может выглядеть так:

```
public class StringUtilsJUnit4CategoriesTest
                                        extends Assert {
  //...
  @Category(JUnit4TestSuite.class)
  @Test
  public void testIsEmpty() { //... }
     //...
  }
```

```
@RunWith(Categories.class)
@Categories.IncludeCategory(JUnit4TestSuite.class)

//настройка категории
@Suite.SuiteClasses({
                    StringUtilsJUnit4CategoriesTest.class})
public class JUnit4TestSuite {
 }
```

Рассмотрим пример использования категорий в тестах

Пусть имеется класс

```
public class JavaTests {
    public int factorial(int n){
        if(n==0)  return 1;
        else { return n*factorial(n-1); }
    }
}
```

Необходимо в нем протестировать метод factorial.

Первый класс с тестами

```
@Category({IShow.class})
public class FooTest {
    JavaTests js;
    @Before
    public void initObject(){ js=new JavaTests(); }
    @Test
    public void thisAlwaysPasses() {
      System.out.println("FooTest-0");
      assertTrue(js.factorial(3)==6);
    }
    @Test
    public void thisAlwaysPasses2() {
      System.out.println("FooTest-2");
      assertTrue(js.factorial(4)==24);
    }
}
```

Другой класс с тестами

```
@Category({IShow.class})
public class FooTest1 {
    JavaTests js;
    @Category({IShow.class})
    @Before
    public void initObject(){ js=new JavaTests();}
    @Category({IShow.class})
    @Test
    public void thisAlwaysPasses() {
      System.out.println("FooTest1-0");
      assertTrue(js.factorial(2)==2);
    }
     @Test
    public void thisAlwaysPasses1() {
      assertTrue(js.factorial(4)==24);
    }
}
```

Запускаемый класс

```
@RunWith(Categories.class)
 @IncludeCategory(IShow.class)
 @SuiteClasses( {FooTest1.class, FooTest.class })
 public class IShow {
 }
```

После запуска

```
java -cp .;d:\distribs\java\junit-4.11.jar;
                d:\distribs\java\hamcrest-all-1.3.jar
          org.junit.runner.JUnitCore IShow
```

получим

```
JUnit version 4.11
I..FooTest1-0
I.FooTest-2
.FooTest-0

Time: 0,009

OK (4 tests)
```

Parameterized — позволяет писать параметризированные тесты.

Для этого в тест-классе объявляется статический метод возвращающий список данных, которые затем будут использованы в качестве аргументов конструктора класса.

```
@RunWith(Parameterized.class)
public class StringUtilsJUnit4ParameterizedTest
                                        extends Assert {
    private final CharSequence testData;
    private final boolean expected;
    public StringUtilsJUnit4ParameterizedTest(
                    final CharSequence testData,
                    final boolean expected ){
        this.testData = testData;
        this.expected = expected;
     }
```

```
@Test
 public void testIsEmpty() {
   final boolean actual = StringUtils.isEmpty(testData);
  assertEquals(expected, actual);
 }
@Parameterized.Parameters
 public static List<Object[]> isEmptyData() {
   return Arrays.asList(new Object[][] { { null, true },
          { "", true }, { " ", false }, { "some string", false }, });
 }
}
```

Theories — чем-то схож с предыдущим, но параметризирует тестовый метод, а не конструктор.

Данные помечаются с помощью **@DataPoints** и **@DataPoint**, тестовый метод — с помощью **@Theory**.

Тест использующий этот функционал будет выглядеть примерно так:

```
@RunWith(Theories.class)
public class StringUtilsJUnit4TheoryTest extends
                                                Assert {
```

```
@DataPoints
public static Object[][] isEmptyData = new Object[][] { {
    "", true }, { " ", false }, { "some string", false } };


@DataPoint
public static Object[] nullData =
                                new Object[] { null, true };
@Theory
public void testEmpty(final Object... testData) {
   final boolean actual =
       StringUtils.isEmpty((CharSequence)testData[0]);
   assertEquals(testData[1], actual);
 }
}
```

Рассмотрим пример. Класс MathFunc оставим без изменений. Класс TestClass имеет вид

```
import org.junit.runner.*;
import org.junit.Assert;
import org.junit.experimental.theories.*;
import org.junit.*;

@RunWith(Theories.class)
public class TestClass extends Assert{

public TestClass() { }

@Test
public void testFactorialNull() {
    MathFunc math = new MathFunc();
    assertTrue(math.factorial() == 1); }
```

```
@DataPoints
public static int[][] Data = new int[][] {{5, 120}};

 @Theory
 public void testFactorialPositive(final int[] obj) {
   MathFunc math = new MathFunc(obj[0]);
   assertTrue(math.factorial() == obj[1]);
  }

 @Test
 public void testPlus() {
  MathFunc math = new MathFunc(45);
  assertTrue(math.plus(123) == 168);
 }
}
```

Запуск теста будет иметь вид:

```
>java –cp .;junit-4.92b2.jar
   org.junit.runner.JUnitCore TestClass
```

На экране

```
Time: 0.015

OK(3 tests)
```

Пример использования @DataPoint

Пусть имеется класс

```
public class JavaTests {
    public int factorial(int n){
        if(n==0)
            return 1;
        else {
            return n*factorial(n-1);
        }
    }
}
```

Тестирующий класс имеет вид:

```
RunWith(Theories.class)
public class FooTest {
    JavaTests js;
    @Before
    public void initObject(){ js=new JavaTests();}

   @DataPoint
    public static int nData=2;

    @Theory
    public void thisAlwaysPasses(int n) {
      assertTrue(js.factorial(n)==2);
    }
}
```

Журналирование Log4j

При небольшом приложении вполне хватет возможностей System.out.println(), но по мере увеличения и усложнения приложения его возможностей уже нехватает.

В частности нельзя (или затруднительно) сделать с помощью System.out.println():

1) управление логгированием: отключить логгирование одних сообщений, и включить логгирование других

2) запись логов в файл, БД и т.п.
3) вывод дополнительной информации о сообщении: время события, место события, контекст
4) форматирование вывода

Для решения этих проблем были созданы системы логгирования.

Две наиболее популярные это **Log4j** от Apache и **Java Logging API** входящее в состав JDK 1.4 и старше.

Рассмотрим пример использования Log4j:

```
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class Test {
  private static Logger logger =
                                  Logger.getLogger(Test.class);

  public static void main(String[] args) throws Exception{
    BasicConfigurator.configure();
    logger.info("Hello world!");
    logger.error("Error!", new Exception("An exception"));
  }
}
```

Для запуска данного приложения нужен пакет log4j-1.2.16.jar.

На экране получим:

```
0 [main] INFO ru.vingrad.log.Test  - Hello world!
0 [main] ERROR ru.vingrad.log.Test  - Error!
java.lang.Exception: An exception
                        at Test.main(Test.java:12)
```

Базовым классом через которое и будет идти все логгирование является **org.apache.log4j.Logger**.

Для получения логгера используется статический метод **getLogger()** куда в качестве параметра надо передать имя этого логгера.

Каждый логгер имеет уникальное имя, и вызов **getLogger()** с одним и тем же именем вернет один и тот же логгер.

Логгер синхронизирован и его можно свободно использовать в многопоточной среде.

Существует корневой логгер, являющийся родительским для всех остальных, получить его можно: **Logger.getRootLogger()**.

Важным фактом является то, что если логгер не сконфигурирован индивидуально, то он будет использовать настройки своего родительского логгера.

Т.е. можно сконфигурировать только корневой логгер, а все логгеры будут использовать его настройки.

Логгер может писать сообщения разных уровней (**org.apache.log4j.Level**).

Уровни упорядочены по старшинству (в порядке убывания): **FATAL, ERROR, WARN, INFO, DEBUG, TRACE**.

Уровни

FATAL, ERROR, WARN предназначены для сообщений об ошибках, разной степени серьезности,

INFO информационные сообщения,

DEBUG - отладочные,

TRACE - очень подробная отладочная информация.

Когда конфигурируется система логгирования то указывается сообщения какого уровня надо писать в лог.

В лог будут писаться сообщения этого уровня и выше, т.е если система настроена на уровень INFO то будут писаться сообщения с уровнями FATAL, ERROR, WARN, INFO. Плюс еще есть спец уровни **ALL** и **OFF**, которые предназначены для включения/выключения записи всех сообщений.

Следующая составляющая системы логгирования это - аппендер (**org.apache.log4j.Appender**).

Аппендер - это класс который занимается непосредственно записью сообщений.

org.apache.log4j.Appender это интерфейс, но у него есть реализации для записи сообщений в консоль, файл, БД, e-mail и т.д.

Так же у аппендера можно установить фильтры (**org.apache.log4j.spi.Filter**) для фильтрации сообщений.

И форматер сообщений (**org.apache.log4j.Layout**) для форматирования выводимых сообщений.

Рассмотрим процесс записи логов:

1. Вызываем logger.info("Hello world!")

2. Происходит проверка уровня логгирования для данного логгера, если данному логгеру не был явно задан уровень, то тогда проверка осуществляется для его родителей.

   Если уровень логгера выше уровня сообщения, то оно отбрасывается и запись его не происходит.

   В нашем случае корневой аппендер сконфигурирован на уровень DEBUG а у сообщения уровень INFO, т.е. уровень логгера ниже уровня сообщения и оно проходит проверку.

3. Далее сообщение передается аппендерам установленным для данного логгера (если есть).

4. Если свойство **additivity** установлено в **true** (по умолчанию оно установлено), то сообщение так же передается родительским логгерам.

5. Аппендер получив сообщение проверяет его установленными фильтрами (если есть). Если сообщение прошло фильтры оно форматируется и записывается в лог.

Рассмотрим конфигурацию логгеров.

Для конфигурирования Log4j используются файлы конфигурации log4j.xml и log4j.properties.

Далее рассмотрим конфигуратор log4j.xml

Создаем в папке с исходниками файл **log4j.xml:**

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
  <!DOCTYPE log4j:configuration
                                    SYSTEM "log4j.dtd">
  <log4j:configuration></log4j:configuration>
```

(файл log4j.dtd можно найти внутри log4j.jar)

Добавим аппендеры. Консоль:

```
<appender name="CONSOLE-DEBUG"
                    class="org.apache.log4j.ConsoleAppender">
    <param name="target" value="System.out"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern"
                    value="%d{ISO8601} [%5p] %m at %l%n"/>
    </layout>
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
      <param name="LevelMin" value="ALL"/>
      <param name="LevelMax" value="INFO"/>
    </filter>
  </appender>
```

CONSOLE-DEBUG имя аппендера, по этому имени он будет упоминаться в этом конфиге.

org.apache.log4j.ConsoleAppender - имя класса аппендера.

<param name="target" value="System.out"/> - устанавливает параметр аппендера который определяет куда писать лог

<layout - форматер.

<filter - фильтр сообщений, в данный аппендер будут писаться все сообщения уровней INFO, DEBUG и TRACE

Аналогично конфигурируется аппендер для ошибок

```
<appender name="CONSOLE-WARN"
            class="org.apache.log4j.ConsoleAppender">
    <param name="target" value="System.err"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
          value="%d{ISO8601} [%5p] %m at %l%n"/>
    </layout>
    <filter
      class="org.apache.log4j.varia.LevelRangeFilter">
      <param name="LevelMin" value="WARN"/>
    </filter>
  </appender>
```

разница в том, ошибки будут писаться в System.err.

Конфигурируем запись в файл

```
<appender name="LOG-FILE-APPENDER"
        class="org.apache.log4j.FileAppender">
  <param name="file" value="app.log"/>
  <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern"
        value="%d{ISO8601} [%5p] %c %m at %l%n"/>
  </layout>
</appender>
```

параметр file задает имя файла.

Теперь сконфигурируем логгеры.

Предположим что используется Hibernate и необходимо видеть только сообщения об ошибках и писать их только в консоль но не файл.

```
<category  name="org.hibernate" additivity="false">
        <priority value="WARN"/>
        <appender-ref ref="CONSOLE-WARN"/>
        <appender-ref ref="CONSOLE-DEBUG"/>
</category>
```

Но в то же самое время, необходимы отладочные сообщения в **org.hibernate.SQL** (где пишутся выполняемые Hibernate SQL запросы).

```
<category name="org.hibernate.SQL">
    <priority value="DEBUG"/>
</category>
```

Для всех остальных категорий включаем уровень DEBUG и запись в консоль и файл

```
<root>
    <priority value="DEBIG"/>
    <appender-ref ref="CONSOLE-WARN"/>
    <appender-ref ref="CONSOLE-DEBUG"/>
    <appender-ref ref="LOG-FILE-APPENDER"/>
</root>
```

Протестировать данный xml файл можно следующим кодом:

```
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class Test {
  private static Logger logger = Logger.getLogger(Test.class);
  public static void main(String[] args) throws Exception {
    logger.debug("Inside main()");
    logger.info("Hello world!");
    logger.error("Error!", new Exception("An exception"));
    Logger hibernateGeneral = Logger.getLogger("org.hibernate");
    hibernateGeneral.debug("Starting Hibernate");
    Logger hibernateSql = Logger.getLogger("org.hibernate.SQL");
    hibernateSql.debug("select * from my table");
    hibernateGeneral.error("Hibernate error");
 }
}
```

Содержимое файла app.log имеет вид:

2011-06-10 01:47:28,968 [DEBUG] Test Inside main() at Test.main(Test.java:9)

2011-06-10 01:47:28,968 [INFO] Test Hello world! at Test.main(Test.java:10)

2011-06-10 01:47:28,968 [ERROR] Test Error! at Test.main(Test.java:11)

java.lang.Exception: An exception

at Test.main(Test.java:11)

Рассмотрим пример:

```
import org.apache.log4j.Logger;
import org.apache.log4j.FileAppender;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.Level;
import java.io.IOException;
public class DemoLog {
  static Logger logger = Logger.getLogger(DemoLog.class);
  public static void main(String[] args) {
     try {
       factorial(9);
       factorial(-3);
     } catch (IllegalArgumentException e) {
       logger.error("negative argument", e);
     }
 }
```

```
public static int factorial(int n) {
  if (n < 0) throw new IllegalArgumentException(
                                        "argument " + n +" less then zero");
     logger.debug("Argument n is " + n);
     int result = 1;
     for (int i = n; i >= 1; i--)
        result *= i;
       logger.info("Result is " + result);
  return result;
}
```

При этом в корне проекта должен находиться конфигурационный файл **"log4j.xml"** со следующим содержимым:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM log4j.dtd">
<log4j:configuration
            xmlns:log4j='http://jakarta.apache.org/log4j/'>
  <appender name="TxtAppender"
                  class="org.apache.log4j.FileAppender">
    <param name="File" value="log.txt" />
    <layout class="org.apache.log4j.SimpleLayout"/>
  </appender>
```

```
  <logger name="app6">
    <level value="debug" />
  </logger>
  <root>
    <appender-ref ref="TxtAppender" />
  </root>
</log4j:configuration>
```

Maven

Жизненный цикл Maven имеет вид:

validate — проверяет корректность метаинформации о проекте

compile — компилирует исходники

test — прогоняет тесты классов из предыдущего шага

package — упаковывает скомпилированые классы в удобноперемещаемый формат (jar или war, к примеру)

integration-test — отправляет упаковынные классы в среду интеграционного тестирования и прогоняет тесты

verify — проверяет корректность пакета и удовлетворение требованиям качества

install — загоняет пакет в локальный репозиторий, откуда он (пекат) будет доступен для использования как зависимость в других проектах

deploy — отправляет пакет на удаленный production сервер, откуда другие разработчики его могут получить и использовать

При этом все шаги последовательны. И если, к примеру, выполнить $ mvn

При этом все шаги последовательны.

И если, к примеру, выполнить $ mvn package, то фактически будут выполнены шаги: validate, compile, test и package.

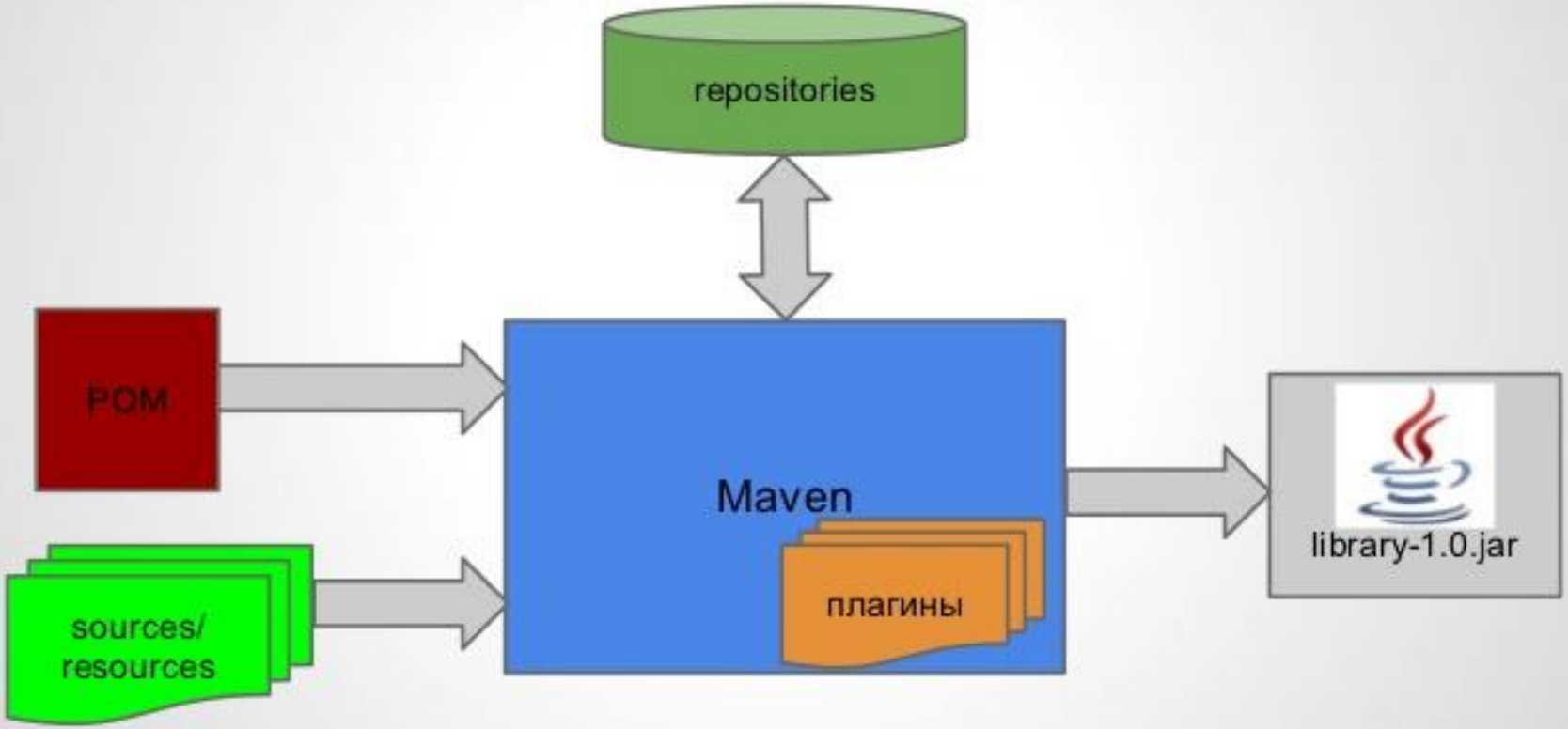
Таким образом если написан код, то можно выполнить mvn test для проверки.

Maven конфигурируется файлом pom.xml, который может содежать блок <dependencies />.

В нем описывается какие библиотеки нужны проекту для полноценного функционирования.

На шаге validate мавен проверяет удовлетворены ли зависимости и если нет, то скачивает из удаленный репозиториев необходимые компоненты в локальный репозиторий.

Как все это работает?



Плагины под Maven

maven-archetype-plugin

Плагин maven-archetype-plugin предназначен для того чтобы по существующему шаблону создавать новые проекты.

Создать проект с помощью maven-archetype-plugin достаточно просто - достаточно набрать:

mvn archetype:generate

maven-compiler-plugin

Компилятор - основной плагин который используется практически во всех проектах. Он доступен по умолчанию, но практически в каждом проекте его приходится переобъявлять т.к. настройки по умолчанию не очень подходящие.

Пример использования:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.0.2</version> <configuration>
  <source>1.6</source>
   <target>1.6</target>
   <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

В этом примере в кофигурации используется версия java 1.6 (source - версия языка на котором написана программа; target - версия java машины которая будет этот код запускать) и указано что кодировка исходного кода программы UTF-8.

maven-surefire-plugin - плагин для запуска тестов

maven-surefire-plugin - плагин который запускает тесты и генерирует отчёты по результатам их выполнения.

По умолчанию отчёты сохраняются в ${basedir}/target/surefire-reports и находятся в двух форматах - txt и xml. Тесты можно писать используя как JUnit так и TestNG.

по умолчанию запускаются все тесты с такими именами "**/Test*.java" - включает все java файлы которые начинаются с "Test" и расположены в поддиректориях.

"**/*Test.java" - включает все java файлы которые заканчиваются на "Test" и расположены в поддиректориях.

"**/*TestCase.java" - включает все java файлы которые заканчиваются на "TestCase" и расположены в поддиректориях.

Пропустить выполнение тестов можно

```
<configuration>
  <skipTests>true</skipTests>
</configuration>
```

или mvn install -DskipTests

чтобы пропустить ещё и компиляцию тестов:

mvn install -Dmaven.test.skip=true

maven-javadoc-plugin

Плагин maven-javadoc-plugin предназначен для того чтобы генерировать докуметацию по исходному коду проекта стандартной утилитой javadoc.

maven-site-plugin

Плагин maven-site-plugin предназначен для того чтобы легко создавать сайт проекта. В директории Вашего проекта наберите

mvn site

И мавен автоматически сгенерирует сайт по нему.

Сгенерированные файлы будут лежать в директории target/site.

maven-checkstyle-plugin

Плагин проверяет стиль и качество исходного кода.

Из наиболее часто используемых и простых проверок:

- наличие комментариев
- размер класса не более N строк
- в конструкции в try-catch, блок catch не пустой.
- не используется System.out.println(.. вместо LOG.error(..

Подключить плагин можно следующим образом:

```
<plugins>
  <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-checkstyle-plugin</artifactId>
     <version>2.7</version>
 </plugin>
.....
```

после этого можно запустить проверку кода:

mvn checkstyle:check

Позволяет создавать собственный набор правил.

findbugs-maven-plugin

findbugs-maven-plugin плагин для автоматического нахождения багов в проекте.

Принцип действия плагина основан на поиске шаблонов ошибок.

Код проекта проверяются на часто встречаемые ошибки, неправильное использование API и конструкций языка.

Пример настройки данного плагина:

```
<reporting>
   <plugins>
      <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>findbugs-maven-plugin</artifactId>
         <version>2.5.2</version>
         <configuration>
            <!--
               Enables analysis which takes more memory but finds more bugs.
               If you run out of memory, changes the value of the effort element
               to 'low'.
            -->
            <effort>Max</effort>
            <!-- Reports all bugs (other values are medium and max) -->
            <threshold>Low</threshold>
            <!-- Produces XML report -->
            <xmlOutput>true</xmlOutput>
         </configuration>
      </plugin>    </plugins> </reporting>
```

```
<build>
   <plugins>
      <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>findbugs-maven-plugin</artifactId>
         <version>2.5.2</version>
         <configuration>
            <!--
               Enables analysis which takes more memory but finds more bugs.
               If you run out of memory, changes the value of the effort element
               to 'Low'.
            -->
            <effort>Max</effort>
            <!-- Reports all bugs (other values are medium and max) -->
            <threshold>Low</threshold>
            <!-- Produces XML report -->
            <xmlOutput>true</xmlOutput>
            <!-- Configures the directory in which the XML report is created -->

<findbugsXmlOutputDirectory>${project.build.directory}/findbugs</findbugsXml
```

```
<!-- Configures the directory in which the XML report is created -->
            <findbugsXmlOutputDirectory>${project.build.directory}/findbugs
           </findbugsXmlOutputDirectory>
         </configuration>
         <executions>
           <!--
           Ensures that FindBugs inspects source code when project is compiled.
            -->
            <execution>
               <id>analyze-compile</id>
               <phase>compile</phase>
               <goals>
                  <goal>check</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins> </build>
```

Создание проекта в Maven

mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes
  -DgroupId=com.mycompany.app
  -DartifactId=my-app

Для создания,например web-приложения

mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-webapp
-DarchetypeArtifactId=maven-archetype-webapp

Файл проекта для web приложения будет

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ua.unicyb.app</groupId>
  <artifactId>webapps</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>webapps Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
```

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
 </dependencies>
```

```
<build>
    <finalName>webapps</finalName>
<plugins>
  <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
       <configuration>
            <!-- id-шник из settings.xml -->
            <server>apache-tomcat</server>
            <url>http://localhost:8080/manager/text</url>
           <path>/mywebapp</path>
         </configuration>
      </plugin>
</plugins>
  </build>

</project>
```

Файл settings.xml имеет вид:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
<servers>
<server>
        <!--
            id это просто идентификатор.
         -->
        <id>apache-tomcat-6.0.32</id>
        <!-- имя пользователя, которое указано в tomcat-users.xml -->
        <username>tomcat</username>
        <!-- пароль из tomcat-users.xml -->
        <password>tomcat</password>
      </server>
    </servers>

</settings>
```

Файл tomcat-user.xml

```
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager-gui"/>
  <role rolename="manager-script"/>
 <role rolename="admin-gui"/>
  <user username="tomcat" password="tomcat" roles="tomcat,
      manager-script,manager-gui,admin-gui"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
</tomcat-users>
```

Java Native Interface

Java Native Interface (JNI) — стандартный механизм для запуска кода, под управлением виртуальной машины Java (JVM), который написан на языках C/C++ или Ассемблера, и скомпонован в виде динамических библиотек, позволяет не использовать статическое связывание.

Это даёт возможность вызова функции C/C++ из программы на Java, и наоборот.

Рассмотрим пример использования JNI.

Имеется следующий Java проект

```
package javaapplication114;
public class TestNativeClass {
    private static int h=30;
        static{
           System.load("d:\\temp\\dll\\javadll.dll");
        }

        private int g;
        public TestNativeClass(int g){
            this.g=g;
        }
       public native int sum(int a,int b);
        public int mul(int a,int b){
            return a*b;
        }
}
```

Свяжем native метод в классе **TestNativeClass** с методом описанным в javadll.dll.

Для этого после компиляции класса **TestNativeClass** используем утилиту javah находящуюся там же где и java.

Эта утилита создаст .h файл, который будет иметь вид:

```
#include <jni.h>
#ifndef _Included_javaapplication114_TestNativeClass
#define _Included_javaapplication114_TestNativeClass
#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT jint JNICALL
  Java_javaapplication114_TestNativeClass_sum(JNIEnv *, jobject, jint,
                                                                  jint);
#ifdef __cplusplus
}
#endif
```

Описываем реализацию метода

```
JNIEXPORT jint JNICALL
        Java_javaapplication114_TestNativeClass_sum(JNIEnv *, jobject, jint,  jint);
```

Имеем:

```
JNIEXPORT jint JNICALL
   Java_javaapplication114_TestNativeClass_sum(JNIEnv * jenv, jobject obj,
                                                                  jint a, jint b)
 {
    jclass javaClass = jenv->GetObjectClass(obj);
//вызываем java метод int mul(int a,int b)
    jmethodID javaFieldID = jenv->GetMethodID(javaClass, "mul", "(II)I");
    jint c=jenv->CallIntMethod(obj, javaFieldID, a, b);
//достаем нестатическое поле private int gc
    jfieldID jf = jenv->GetFieldID(javaClass, "g","I");
    jint g = jenv->GetIntField(obj, jf);
//достаем статическое поле  private static int h=30
    jfieldID jfield = jenv->GetStaticFieldID(javaClass, "h", "I");
    jint h = jenv->GetStaticIntField(javaClass, jfield);
    jint d = a + c + g+h;
    return d;}
```

Для компиляции C++ проекта необходимо подключитиь директории (с ключем **– I**)

c:\Program Files\Java\jdk1.8.0\include\

а также

c:\Program Files\Java\jdk1.8.0\include\win32\