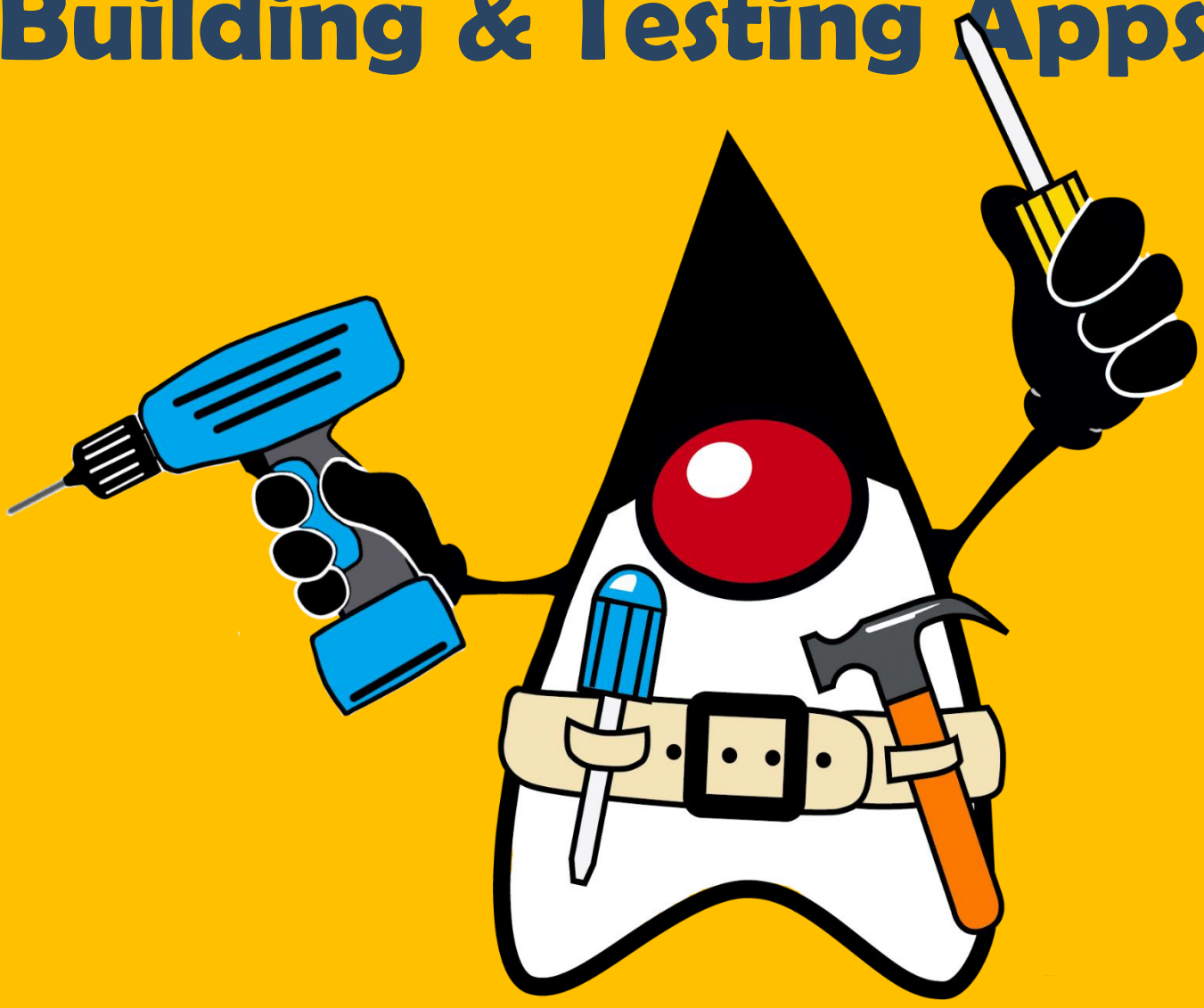
Basic Java Tools for Building & Testing Apps



Maven

JUnit

Gradle

Mockito

Содержание

Введение. Сборка проекта

Одним из важных аспектов подготовки нового релиза приложения являются не только сборка проекта, но также и его тестирование. Здесь следует отметить, что грамотно определенный процесс тестирования приложения позволяет выполнить не только функциональное тестирование, но также модульное и интеграционное тестирование.

Автоматизация процесса сборки программного продукта связана с разработкой различных скриптов для выполнения таких действий, как :

- компиляция исходного кода в бинарный;
- сборка бинарного кода;
- разворачивание программы на сервере (удаленном компьютере);
- оформление сопроводительной документации или описание изменений.

Процесс разработки проекта группой программистов включает сборку очередной версии проекта, после которого необходимо выполнить тестирование этой версии. Чтобы собрать очередную версию, ответственный за сборку должен получить последние версии файлов, обновить номер версии (релиза), собрать проект и, если все пройдет успешно, выложить готовый проект для тестирования в определенное место. При этом необходимо использовать определенные библиотеки. Соответственно переход на новую версию программного продукта или одной из библиотек может вызвать множество проблем: сборка новой версии может оказаться неработоспособной и придется откатываться на старую версию и т.п.

Таким образом, автоматизация процесса сборки проекта становится неотъемлемой частью процесса разработки. С развитием интегрированных средств разработки (IDE Eclipse, NetBeans, IntelliJ IDEA и т.п.) стало ясно, что пакетные файлы для автоматизации сборки уже не соответствуют современным требованиям. Появились новые системы автоматизации сборки Apache Ant, Apache Maven.

Apache Ant (http://ant.apache.org) сталлогическим продолжением *make*, схожий по принципу работы инструмент, основной задачей которого была обеспечить автоматизацию процесса сборки Java приложений. Ant - это императивная командная система, созданная для кроссплатформенного применения. Изначально *ant* разрабатывался для сборки и компановки java-проектов.

Для java-разработчиков большой проблемой применения пакетных файлов было то, что они сильно завязаны на команды операционной системы (ОС). В случае, если необходимо было обеспечить сборку проекта в разных операционных системах, то нужно было использовать не только разные наборы и параметры команд, но и формат командных файлов. Разработка

собственного командного файла сборки под каждую платформу – это не лучший выбор для кроссплатформенных приложений. В результате *Apache Ant* был спроектирован таким образом, что часто применяемые при сборке команды ОС обернуты внутренними командами ant, а скрипты сборки описываются в формате XML.

Широкое распространение получила декларативная система автоматизации сборки **Apache Maven** согласно описанию проектного *pom.xml* файла формата XML. В файлах проекта *pom.xml* содержатся не отдельные команды, а описание проекта. Все задачи по обработке файлов *maven* выполняет с использованием плагинов.

В настоящее время *maven* развился в специализированную комплексную систему управления сложным процессом создания программного обеспечения и представляет собой обобщенную систему управления разработкой с огромным количеством дополнительных возможностей, применяемых в большинстве сценариев разработки ПО. Многие среды разработки приложений для языка Java обеспечивают интеграцию с этой системой.

Неоспоримым преимуществом maven является автоматическое управление зависимостями, хорошая структурированность проектов и отсутствие скриптов сборки как таковых, а следовательно проблем с ними. К недостаткам этой системы обычно относят сложности в изучении и трудность диагностики проблем при сборке. Также следует отметить сложности в поиске нужных maven плагинов и их настройке.

В связи с большим объемом информации по Apache Maven на сайте представлено несколько страниц, где можно ближе познакомиться с общим описанием данного программного продукта и его возможностями, а также увидеть многочисленные примеры использования данной системы.

Фреймворк Apache Maven

Apache Maven предназначен для автоматизации процесса сборки проектов на основе описания их структуры в файле на языке POM (Project Object Model), который является подмножеством формата XML. *maven* позволяет выполнять компиляцию кодов, создавать дистрибутив программы, архивные файлы jar/war и генерировать документацию. Простые проекты **maven** может собрать в командной строке. Название программы **maven** вышло из языка идиш, смысл которого можно выразить как «собиратель знания».

В отличие от ant с императивной сборкой проекта, **maven** обеспечивает декларативную сборку проекта. То есть, в файле описания проекта содержатся не отдельные команды выполнения, а спецификация проекта. Все задачи по обработке файлов *maven* выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Если описать *maven* на одной странице сайта, да еще привести примеры использования, то содержимое будет «нечитабельным» - это слишком большой объём информации для представления далеко не всех его возможностей. Поэтому описание разнесено по нескольким страницам. На этой странице представлено общее описание *maven*. На странице Примеры проектов maven описано использование *maven* для разнотипных проектов. Отдельными страницами представлено «применение maven в IDE Eclipse» (в разработке) и «плагины maven» (в разработке).

Инсталляция maven

Последнюю версию *maven* можно скачать в виде zip-архива со страницы загрузки на официальном сайте. После этого необходимо выполнить несколько шагов :

- распаковать архив в инсталляционную директорию. Например в директорию C:\maven-x.y.z в Windows или /opt/maven-x.y.z в Linux
- установить переменную окружения M2_HOME :
  - в Windows кликните правой кнопкой мыши на «Мой компьютер» и откройте окно Свойства/«Дополнительные параметры»/«Переменные среды»/«Системные переменные», в котором добавьте «M2_HOME» = "C:\maven-x.y.z\";
  - в Linux можно добавить строку «export M2_HOME=/opt/maven-x.y.z» в файл /etc/profile;
- Внесите изменения в переменную окружения PATH :
  - в Windows в переменную PATH добавьте строку %M2_HOME%\bin;
  - в Linux можно добавить строку «export PATH=$PATH:$M2_HOME/bin» в файл /etc/profile;

Чтобы убедиться, что **maven** установлен, необходимо в командной строке ввести следующую команду :

```
mvn --version
```

Должна появиться информация о версиях **maven**, *jre* и операционной системе типа :

```
```

```
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-
10T19:41:47+03:00)
Maven home: C:\apache-maven-3.3.9
Java version: 1.7.0_79, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_79\jre
Default locale: ru_RU, platform encoding: Cp1251
OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"
```

При инсталляции **maven**'а будет создан локальный [репозиторий](#) в вашей личной папке ${user.home}\.m2\repository. После этого можно считать, что **maven** готов к работе и можно приступать к созданию проектов сборки приложений.

Одной из привлекательных особенностей *maven*'а является справка online, работоспособность которой можно проверить после инсталляции. К примеру справку по фазе компиляции можно получить следующей командой :

```
mvn help:describe -Dcmd=compile
```

В результате Вы увидите следующую справочную информацию :

```
[INFO] 'compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

It is a part of the lifecycle for the POM packaging 'jar'.
This lifecycle includes the following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: \
        org.apache.maven.plugins:maven-resources-plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: \
        org.apache.maven.plugins:maven-resources-plugin:2.6:testResources
* test-compile: \
        org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.742 s
[INFO] Finished at: 2016-09-19T22:41:26+04:00
[INFO] Final Memory: 7M/18M
[INFO] ------------------------------------------------------------------------
```

Репозитории проекта, repositories

Под репозиторием (*repository*) понимается, как правило, внешний центральный репозиторий, в котором собрано огромное количество наиболее популярных и востребованных библиотек, и локальный репозиторий, в котором хранятся копии используемых ранее библиотек.

Дополнительные репозитории, необходимые для сборки проекта, перечисляются в секции *<repositories>* проектного файла *pom.xml* :

```
<project>
    ...
    <repositories>
        <repository>
            <id>repo1.maven.org</id>
            <url>http://repo1.maven.org/maven2</url>
        </repository>
    </repositories>
    ...
</project>
```

Можно создать и подключать к проектам свой репозиторий, содержимое которого можно полностью контролировать и сделать его доступным для ограниченного количества пользователей. Доступ к содержимому репозитория можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен из вне. Существуют несколько реализаций серверов - репозиториев maven; к наиболее известным относятся artifactory, continuum, nexus.

Таким образом, репозиторий - это место, где хранятся файлы jar, pom, javadoc, исходники и т.д. В проекте могут быть использованы :

- центральный репозиторий, доступный на чтение для всех пользователей в интернете;
- внутренний «корпоративный» репозиторий - дополнительный репозиторий группы разработчиков;
- локальный репозиторий, по умолчанию расположен в ${user.home}/.m2/repository - персональный для каждого пользователя.

Для добавления, к примеру, библиотеки *carousel-lib.jar* в локальный репозиторий можно использовать команду *mvn install* (команда должна быть однострочной) :

```
mvn install:install-file \
    -Dfile=${FILE_PATH}/carousel-lib.jar \
    -DgroupId=ru.carousel \
    -DartifactId=carousel-lib \
    -Dversion=1.0 \
    -Dpackaging=jar \
    -DgeneratePom=true
```

В локальном репозитории «.m2» *maven* создаст директорию ru/carousel, в которой разместит данную библиотеку и создаст к ней описание в виде pom.xml.

Репозиторий можно также разместить внутри проекта. Описания процесса создания и размещения репозитория внутри проекта с примером можно прочитать здесь.

Собстенные наработки можно подключить как системную зависимость.

Терминология maven

В **maven** используется свой набор терминов и понятий. Ключевым понятием *maven* является артефакт (*artifact*) — это, по сути, любая библиотека, хранящаяся в репозитории, к которой можно отнести зависимость или плагин.

Зависимости (*dependencies*) представляют собой библиотеки, которые непосредственно используются в проекте для компиляции или тестирования кода.

При сборке проекта или для каких-то других целей (deploy, создание файлов проекта для Eclipse и др.) *maven* использует плагины (*plugin*).

Еще одним важным понятием **maven** проекта является архетип (*archetype*) - это некая стандартная компоновка каталогов и файлов в проектах различного типа (web, maven, swt/swing-проекты и прочие). Иными словами *maven* знает, как построить структуру проекта в соответствии с его архетипом.

Архетипы maven

Количество архетипов у *maven*'а огромно, «на разный вкус». Как правильно выбрать нужный, чтобы создать архитектуру будущего проекта? Просматривать в консоли не очень удобно, тем более что их количество переваливает за 1500 (к примеру для версии maven 3.3.9 на моем компьютере их 1665). Поэтому можно скачать их в отдельный файл, а потом со всем этим хозяйством разбираться. Для этого необходимо выполнить следующую команду :

```
mvn archetype:generate > archetypes.txt
```

В результате в файле archetypes.txt можно увидеть, что-то подобное

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) >
                    generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) <
                    generate-sources @ standalone-pom <<<
[INFO]
[INFO] maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart \
       (org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
1: remote -> am.ik.archetype:maven-reactjs-blank-archetype
                                    (Blank Project for React.js)
2: remote -> am.ik.archetype:msgpack-rpc-jersey-blank-archetype
                                    (Blank Project for Spring Boot + Jersey)
3: remote -> am.ik.archetype:mvc-1.0-blank-archetype
                                    (MVC 1.0 Blank Project)
4: remote -> am.ik.archetype:spring-boot-blank-archetype
                                    (Blank Project for Spring Boot)
5: remote -> am.ik.archetype:spring-boot-docker-blank-archetype
                                    (Docker Blank Project for Spring Boot)

...
```

При выполнении данной команды *maven*, после скачивания информации в файл, ожидает поступления команды пользователя. Т.е., находится в ожидании интерактивного ввода команд по созданию проекта определенного типа. Если файл уже создан, то прервите выполнение команды двойным нажатием клавишам Ctrl+C.

Для создания простенького *maven* проекта «carousel» (карусель) необходимо выполнить следующую команду :

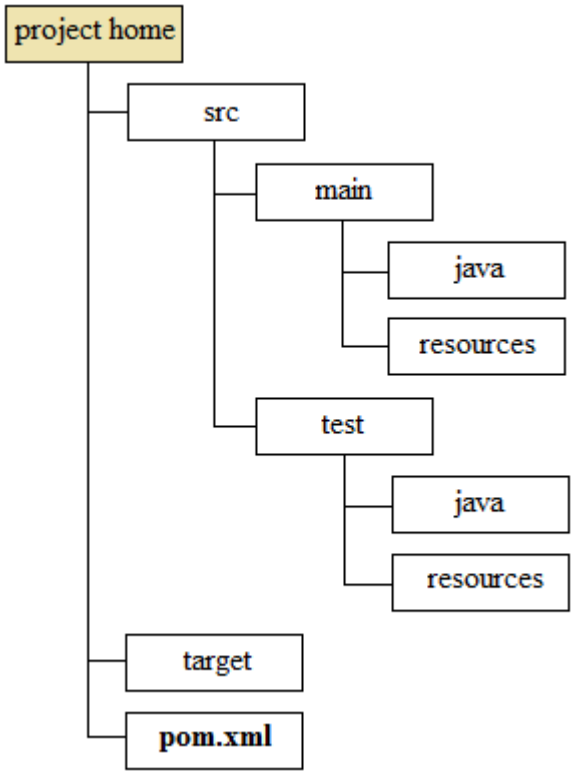
```
mvn archetype:generate \
    -DgroupId=ru.carousel \
    -DartifactId=carousel \
```

```
    -Dversion=1.0-SNAPSHOT \
    -DarchetypeArtifactId=maven-archetype-quickstart
```

На странице Примеры maven проектов подробно описываются разнотипные **maven** проекты - проект консольного приложения без зависимостей, приложение с графическим интерфейсом и с зависимостями, web-приложение с фреймворком.

Архитектура простого maven проекта

Следующая структура показывает директории простого *maven* проекта.



Проектный файл *pom.xml* располагается в корне каталога.

- src: исходные файлы;
- src/main: исходные коды проекта;
- src/main/java: исходные java-файлы;
- src/main/resources: ресурсные файлы, которые используются при компиляции или исполнении, например properties-файлы;
- src/test: исходные файлы для организации тестирования;
- src/test/java: JUnit-тест-задания для автоматического тестирования;
- target: создаваемые в процессе работы maven'а файлы для сборки проекта.

В зависимости от типа приложения (консольное, с интерфейсом, web, gwt и т.д.) структура может отличаться. В директории *target* **maven** собирает проект (jar/war).

На официальном сайте **Apache Maven Project** можно получить дополнительную информацию об архетипах (Introduction to Archetypes).

Жизненный цикл maven проекта

Жизненный цикл **maven** проекта – это чётко определённая последовательность фаз.
Когда *maven* начинает сборку проекта, он проходит через определённую последовательность фаз, выполняя задачи, указанные в каждой из фаз. *Maven* имеет 3 стандартных жизненных цикла :

- clean — жизненный цикл для очистки проекта;

- default — основной жизненный цикл;
- site — жизненный цикл генерации проектной документации.

Каждый из этих циклов имеет фазы pre и post. Они могут быть использованы для регистрации задач, которые должны быть запущены перед и после указанной фазы.

Фазы жизненного цикла clean

- pre-clean;
- clean;
- post-clean.

Фазы жизненного цикла default

- validate - выполнение проверки, является ли структура проекта полной и правильной;
- generate-sources - включение исходного кода в фазу;
- process-sources - подготовка/обработка исходного кода; например, фильтрация определенных значений;
- generate-resources - генерирование ресурсов, которые должны быть включены в пакет;
- process-resources - копирование ресурсов в указанную директорию (перед упаковкой);
- compile - компиляция исходных кодов проекта;
- process-test-sources - обработка исходных кодов тестов;
- process-test-resources - обработка ресурсов для тестов;
- test-compile - компиляция исходных кодов тестов;
- test - собранный код тестируется, используя приемлемый фреймворк типа JUnit;
- package - упаковка откомпилированных классов и прочих ресурсов в дистрибутивный формат;
- integration-test - программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта;
- install - установка программного обеспечения в maven-репозиторий, чтобы сделать его доступным для других проектов;
- deploy - стабильная версия программного обеспечения копируется в удаленный maven-репозиторий, чтобы сделать его доступным для других пользователей и проектов;

Фазы жизненного цикла site

- pre-site;
- site;
- post-site;
- site-deploy;

Стандартные жизненные циклы могут быть дополнены функционалом с помощью *maven-плагинов*. Плагины позволяют вставлять в стандартный цикл новые шаги (например, распределение на сервер приложений) или расширять существующие шаги.

Порядок выполнения команд **maven** проекта зависит от порядка вызова целей и фаз. Следующая команда

```
mvn clean dependency:copy-dependencies package
```

выполнит фазу *clean*, после этого будет выполнена задача *dependency:copy-dependencies*, после чего будет выполнена фаза *package*. Аргументы *clean* и *package* являются фазами сборки, *dependency:copy-dependencies* является задачей.

Зависимости, dependency

Зависимость, эта связь, которая говорит, что для некоторых фаз жизненного цикла *maven* проекта, требуются некоторые артефакты. Зависимости проекта описываются в

секции *<dependencies>* файла *pom.xml*. Для каждого используемого в проекте артефакта необходимо указать GAV (**g**roupId, **a**rtifactId, **v**ersion), где

- **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
- **artifactId** - идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
- **version** - версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс SNAPSHOT. Например 1.0-SNAPSHOT.

Как правило информации GAV достаточно *maven*'у, для поиска указанного артефакта в репозиториях. Пример описания зависимости библиотеки JDBC для работы с БД Oracle.

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0.4</version>
</dependency>
```

Но иногда при описании зависимости требуется использовать необязательный параметр <classifier>. Следующий пример демонстрирует описание зависимости библиотеки *json-lib-2.4-jdk15.jar* с параметром *classifier*.

```
<dependency>
    <groupId>net.sf.json-lib</groupId>
    <artifactId>json-lib</artifactId>
    <version>2.4</version>
    <classifier>jdk15</classifier>
</dependency>
```

Более подробная информация о зависимостях и областях их действия, а также о способе построения транзитивных зависимостей в виде дерева представлена на странице dependency в maven-проекте.

Плагины, plugins

Maven базируется на plugin-архитектуре, которая позволяет использовать плагины для различных задач (test, compile, build, deploy и т.п). Иными словами, *maven* запускает определенные плагины, которые выполняют всю работу. То есть, если мы хотим научить *maven* особенным сборкам проекта, то необходимо добавить в *pom.xml* указание на запуск нужного плагина в нужную фазу и с нужными параметрами. Это возможно за счет того, что информация поступает плагину через стандартный вход, а результаты пишутся в его стандартный выход.

Количество доступных плагинов очень велико и включает разнотипные плагины, позволяющие непосредственно из *maven* запускать web-приложение для тестирования его в браузере, генерировать Web Services. Главной задачей разработчика в этой ситуации является найти и применить наиболее подходящий набор плагинов.

В простейшем случае запустить плагин просто - для этого необходимо выполнить команду в определенном формате. Например, чтобы запустить плагин «maven-checkstyle-plugin» (artifactId) с groupId равным «org.apache.maven.plugins» необходимо выполнить следующую команду :

```
mvn org.apache.maven.plugins:maven-checkstyle-plugin:check
```

Целью (goal) выполнения данного плагина является проверка "check". Можно запустить в более краткой форме :

```
mvn maven-checkstyle-plugin:check
```

Объявление плагина в проекте похоже на объявление зависимости. Плагины также идентифицируется с помощью GAV (groupId, artifactId, version). Например:

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>2.6</version>
    </plugin>
</plugins>
```

Объявление плагина в *pom.xml* позволяет зафиксировать версию плагина, задать ему необходимые параметры, определить конфигурационные параметры, привязать к фазам.

Что касается списка конфигурационных переменных плагина, то его легко можно найти на сайте *maven*. К примеру, для *maven-compiler-plugin*, на странице Apache Maven Project можно увидеть перечень всех переменных, управляющих плагином.

Разные плагины вызываются *maven'ом* на разных стадиях жизненного цикла. Так проект, формирующий настольное java-приложение с использованием библиотек swing или swt, имеет стадии жизненного цикла отличные от тех, что характерны для разработке enterprise application (ear). Еак например, когда выполняется команда «mvn test», инициируется целый набор шагов в жизненном цикле проекта: «process-resources”, «compile», «process-classes», «process-test-resources», «test-compile», «test». Упоминания этих фаз отражаются в выводимых maven-ом сообщениях :

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building carousel 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) \
           @ carousel ---
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) \
           @ carousel
[INFO] --- maven-resources-plugin:2.6:testResources \
           (default-testResources) @ carousel ---
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) \
           @ carousel ---
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) \
           @ carousel ---
[INFO] Surefire report directory: \
           E:\maven.projects\carousel\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running ru.carousel.AppTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.042 s
[INFO] Finished at: 2016-09-24T12:33:45+04:00
[INFO] Final Memory: 7M/18M
[INFO] ------------------------------------------------------------------------
```

В каждой фазе жизненного цикла проекта вызывается определенный плагин (jar-библиотека), который включает некоторое количество целей (goal). Например, плагин «maven-compiler-plugin» содержит две цели: «compiler:compile» для компиляции основного исходного кода проекта и «compiler:testCompile» для компиляции тестов. Формально, список фаз можно изменять, хотя такая ситуация случается крайне редко.

В проектном файле *pom.xml* можно настроить для каждого из плагинов жизненного цикла набор конфигурационных переменных, например :

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <verbose>true</verbose>
        <executable>
              C:/Program_Files/Java/jdk1.7.0_67/bin/javac.exe
        </executable>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

В случае необходимости выполнения нестандартных действий в определенной фазе, например, на стадии генерации исходников «generate-sources», можно добавить вызов соответствующего плагина в файле *pom.xml* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>имя-плагина</artifactId>
  <executions>
    <execution>
      <id>customTask</id>
      <phase>generate-sources</phase>
      <goals>
         <goal>pluginGoal</goal>
      </goals>
...
```

Самое важное в данном случае – это определить для плагина наименование фазы «execution/phase», в которую нужно встроить вызов цели плагина «goal». Фаза «generate-sources» располагается перед вызовом фазы compile и очень удобна для генерирования части исходных кодов проекта.

Описание различных плагинов представлено на странице Maven плагины для сборки проекта.

Основные исполняемые цели, goal

Использование **maven** часто сводится к выполнению одной из команды следующего набора, которые можно назвать целями (по аналогии с другими системами сборки типа ant, make) :

- validate — проверка корректности метаинформации о проекте;
- compile — компилиляция исходников;
- test — прогонка тестов классов;

- package — упаковка скомпилированнных классов в заданный формат (jar или war, к примеру);
- integration-test — отправка упакованных классов в среду интеграционного тестирования и прогонка тестов;
- verify — проверка корректности пакета и удовлетворение требованиям качества;
- install — отправка пакета в локальный репозиторий, где он будет доступен для использования как зависимость в других проектах;
- deploy — отправка пакета на удаленный production сервер, где доступ к нему будет открыт другим разработчикам.

В общем случае для выполнения команды *maven* необходимо выполнить следующий код : «mvn цель». В качестве параметров указываются не только имена фаз, но и имена и цели плагинов в формате «mvn плагин:цель». Например, вызов фазы цикла «mvn clean» эквивалентен вызову плагина «mvn clean:clean».

Секция свойств maven проекта, properties

Отдельные настройки проекта можно определить в переменных. Это может быть связанно, к примеру, с тем, что требуется использовать семейство библиотек определенной версии. Для этого в проектном файле используется секция <properties>, в которой объявляются переменные. Обращение к переменной выглядит следующим образом : ${имя переменной}. Пример описания свойств проекта и их использование :

```
<properties>
    <junit.version>4.11</junit.version>
    <maven.compiler.source>1.4</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
</properties>

…
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
…
<build>
    <finalName>${project.artifactId}</finalName>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
            <source>${maven.compiler.source}</source>
            <target>${maven.compiler.target}</target>
        </configuration>
    </plugin>
</build>
…
```

Кодировка maven проекта

При выполнении отдельных команд *maven*, связанных с копированием ресурсов или компиляцией, могут «выплыть» предупреждения о кодировке :

```
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ ...
[WARNING] Using platform encoding (Cp1251 actually) to copy filtered
                            resources, i.e. build is platform dependent!

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ ...
```

```
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1251,
                                        i.e. build is platform dependent!
```

Чтобы обойти эти сообщения, необходимо включить в секцию <properties> следующий код с указанием требуемой кодировки :

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Для просмотра свойств проекта можно использовать плагин «maven-echo-plugin» :

```
<plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>groovy-maven-plugin</artifactId>
    <version>2.0</version>
    <executions>
        <execution>
            <phase>validate</phase>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                <source>
                    log.info('JUnit версия : {0}', junit.version)
                </source>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Проектный файл, pom.xml

Структура проекта описывается в файле **pom.xml**, который должен находиться в корневой папке проекта. Содержимое проектного файла имеет следующий вид :

```
<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/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!-- GAV параметры описания проекта -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <packaging>...</packaging>
    <version>...</version>

    <!-- Секция свойств -->
    <properties>
        . . .
    </properties>

    <!-- Секция репозиториев -->
    <repositories>
        . . .
    </repositories>

    <!-- Секция зависимостей -->
    <dependencies>
        . . .
```

```
    </dependencies>

    <!-- Секция сборки -->
    <build>
        <finalName>projectName</finalName>
        <sourceDirectory>${basedir}/src/java</sourceDirectory>
        <outputDirectory>${basedir}/targetDir</outputDirectory>
        <resources>
            <resource>
                <directory>${basedir}/src/java/resources</directory>
                <includes>
                    <include>**/*.properties</include>
                </includes>
            </resource>
        </resources>
        <plugins>
            . . .
        </plugins>
    </build>
</project>
```

Не все секции могут присутствовать в описании pom.xml. Так секции properties и repositories часто не используются. Параметры GAV проекта являются обязательными. Выше на странице было представлено описание использования различных секций. Здесь рассмотрим только секцию <build>.

Секция build

Секция <build> также не является обязательной, т. к. существует значение по умолчанию. Данная секция содержит информацию по самой сборке, т.е. где находятся исходные файлы, файлы ресурсов, какие плагины используются.

- <finalName> - наименование результирующего файла сборки (jar, war, ear..), который создаётся в фазе package. Значение по умолчанию — «artifactId-version»;
- <sourceDirectory> - определение месторасположения файлов с исходным кодом. По умолчанию файлы располагаются в директории «${basedir}/src/main/java», но можно определить и в другом месте;
- <outputDirectory> - определение месторасположения директории, куда компилятор будет сохранять результаты компиляции - *.class файлы. По умолчанию определено значение «target/classes»;
- <resources> и вложенные в неё тэги <resource> определяют местоположение файлов ресурсов. Ресурсы, в отличие от файлов исходного кода, при сборке просто копируются в директорию, значение по умолчанию которой равно «src/main/resources».

Тестирование проекта

Maven позволяет запускать JUnit case приложения на тестирование. Для этого следут выполнить команду "mvn test". Отдельные команды maven, например "mvn verify", автоматически запускают тесты приложения. Тестирование можно запретить на уровне выполнения команды или в секции "properties" файла pom.xml. Подробнее информация о тестировании с использованием maven представлена здесь.

Предопределённые переменные maven

При описании проекта в pom-файле можно использовать предопределенные переменные. Их можно условно разделить на несколько групп :

- Встроенные свойства проекта :
  - ${basedir} - корневой каталог проекта, где располагается pom.xml;
  - ${version} - версия артефакта; можно использовать ${project.version} или ${pom.version};

- Свойства проекта. На свойства можно ссылаться с помощью префиксов «project» или «pom» :
  - ${project.build.directory} - «target» директория (можно ${pom.build.directory});
  - ${project.build.outputDirectory} - путь к директории, куда компилятор складывает файлы (по умолчанию «target/classes»);
  - ${project.name} - наименование проекта (можно ${pom.name});
  - ${project.version} - версия проекта (можно ${pom.version}).
- Настройки. Доступ к свойствам settings.xml можно получить с помощью префикса settings
  - ${settings.localRepository} путь к локальному репозиторию

Maven зависимости, dependency

Редко когда какой-либо проект обходится без дополнительных библиотек. Как правило, используемые в проекте библиотеки необходимо включить в сборку, если это не проект OSGi или WEB (хотя и для них зачастую приходится включать в проект отдельные библиотеки). Для решения данной задачи в maven-проекте необходимо использовать зависимость **dependency**, устанавливаемые в файле pom.xml, где для каждого используемого в проекте артефакта необходимо указать :

- параметры GAV (**g**roupId, **a**rtifactId, **v**ersion) и, в отдельных случаях, «необязательный» классификатор classifier;
- области действия зависимостей scope (compile, provided, runtime, test, system, import);
- месторасположение зависимости (для области действия зависимости system).

Параметры GAV

- **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
- **artifactId** - идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
- **version** - версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс SNAPSHOT. Например 1.0.0-SNAPSHOT.

Значения идентификаторов *groupId* и *artifactId* подключаемых библиотек практически всегда можно найти на сайте www.mvnrepository.com. Если найти требуемую библиотеку в этом репозитории не удается, то можно использовать дополнительный репозиторий http://repo1.maven.org/maven2.

Структура файла pom.xml и описание секции подключения к проекту репозитория представлены на главной странице фреймворка maven.

Объявление зависимостей заключено в секции <dependencies>...</dependencies>. Количество зависимостей не ограничено. В следующем примере представлено объявление зависимости библиотеки JSON, в которой используется классификатор *classifier* (в противном случае библиотека не будет найдена в центральном репозитории) :

```
<dependencies>
    <dependency>
        <groupId>net.sf.json-lib</groupId>
        <artifactId>json-lib</artifactId>
        <version>2.4</version>
        <classifier>jdk15</classifier>
    </dependency>
<dependencies>
```

Классификатор classifier

Классификатор **classifier** используется в тех случаях, когда деление артефакта по версиям является недостаточным. К примеру, определенная библиотека (артефакт) может быть

использована только с определенной JDK (VM), либо разработана под windows или linux. Определять этим библиотекам различные версии – идеологически не верно. Но вот использованием разных классификаторов можно решить данную проблему.

Значение *classifier* добавляется в конец наименования файла артефакта после его версии перед расширением. Для представленного выше примера полное наименование файла имеет следующий вид : json-lib-2.4-jdk15.jar.

Расположение артефакта в репозитории

В maven-мире «оперируют», как правило, артефактами. Это относится и к создаваемому разработчиком проекту. Когда выполняется сборка проекта, то формируется наименование файла, в котором присутствуют основные параметры GAV. После сборки этот артефакт готов к установке как в локальный репозиторий для использования в других проектах, так и для распространения в public-репозитории. Помните, что в начале файла pom.xml указываются параметры GAV артефакта :

```
<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>com.examples</groupId>
    <artifactId>example1</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
  ...
</project>
```

Формально координата артефакта представляет четыре слова, разделенные знаком двоеточия в следующем порядке groupId:artifactId:packaging:version.

Полный путь, по которому находится файл артефакта в локальном репозитории, использует указанные выше четыре характеристики. В нашем примере для зависимости JSON это будет "HOME_PATH/.m2/repository/net/sf/json-lib/json-lib/2.4/json-lib-2.4-jdk15.jar". Параметру groupId соответствует директория (net/sf/json-lib) внутри репозитория (/.m2/repository). Затем идет поддиректория с artifactId (json-lib), внутри которой располагается поддиректория с версией (2.4). В последней располагается сам файл, в названии которого присутствуют все параметры GAV, а расширение файла согласуется с параметром *packaging*.

Здесь следует заметить, что правило, при котором «расширение файла с артефактом соответствует его packaging» не всегда верно. К примеру, те, кто знаком с разработкой enterprise приложений, включающих бизнес-логику в виде ejb-модулей и интерфейса в виде war-модулей, знают, что модули ejb-внешне представляют собой обычный архивный файл с расширением jar, хотя в теге *packaging* определено значение ejb.

В каталоге артефакта, помимо самого файла, хранятся связанные с ним файлы с расширениями *.pom, *.sha1 и *.md5. Файл *.pom содержит полное описание сборки артефакта, а в файлах с расширениями sha1, md5 хранятся соответствующие значения MessageDidgest, полученные при загрузке артефакта в локальный репозиторий. Если исходный файл в ходе загрузки по открытым каналам Internet получил повреждения, то вычисленное значения sha1 и md5 будут отличаться от загруженного значения. А, следовательно, *maven* должен отвергнуть такой артефакт и попытаться загрузить его из другого репозитория.

Область действия зависимости, scope

Область действия scope определяет этап жизненного цикла проекта, в котором эта зависимость будет использоваться.

test
Если зависимость junit имеет область действия *test*, то эта зависимость будет использована maven'ом при выполнении компиляции той части проекта, которая содержит тесты, а также при запуске тестов на выполнение и построении отчета с результатами тестирования кода. Попытка сослаться на какой-либо класс или функцию библиотеки *junit* в основной части приложения (каталог src/main) вызовет ошибку.

compile
К наиболее часто используемой зависимости относится *compile* (используется по умолчанию). Т.е. *dependency*, помеченная как *compile*, или для которой не указано *scope*, будет доступна как для компиляции основного приложения и его тестов, так и на стадиях запуска основного приложения или тестов. Чтобы инициировать запуск тестов из управляемого maven-проекта можно выполнив команду "mvn test", а для запуска приложения используется плагин exec.

provided
Область действия зависимости *provided* аналогична *compile*, за исключением того, что артефакт используется на этапе компиляции и тестирования, а в сборку не включается. Предполагается, что среда исполнения (JDK или WEB-контейнер) предоставят данный артефакт во время выполнения программы. Наглядным примером подобных артефактов являются такие библиотеки, как hibernate или jsf, которые необходимы на этапе разработки приложения.

runtime
Область действия зависимости *runtime* не нужна для компиляции проекта и используется только на стадии выполнения приложения.

system
Область действия зависимости *system* аналогична provided за исключением того, что содержащий зависимость артефакт указывается явно в виде абсолютного пути к файлу, определенному в теге *systemPath*. Обычно к таким артефактам относятся собственные наработки, и искать их в центральном репозитории, куда Вы его не размещали, не имеет смысла :

```
<dependencies>
    <dependency>
        <groupId>ru.carousel</groupId>
        <artifactId>carousel-lib</artifactId>
        <version>1.0</version>
        <scope>system</scope>
        <systemPath>
               /projects/libs/carousel-lib.jar
        </systemPath>
    </dependency>
</dependencies>
```

Версия SNAPSHOT

При определении версии релиза можно использовать **SNAPSHOT**, который будет свидетельствовать о том, что данный артефакт находится в процессе разработки и в него вносятся постоянные изменения, например делается bugfixing или дорабатывается функционал. В этом случае код и функциональность артефакта последней сборки в репозитории могут не соответствовать реальному положению дел. Таким образом нужно четко отделять стабильные версии артефактов от не стабильных. Связываясь с нестабильными артефактами нужно быть готовыми к тому, что их поведение может измениться и наш проект, использующий такой артефакт, может вызывать исключения. Следовательно, нужно определиться с вопросом: нужно ли обновлять из репозитория артефакт, ведь его номер формально остался неизменным.

Если версия модуля определяется как SNAPSHOT (версия 2.0.0-SNAPSHOT), то maven будет либо пересобирать его каждый раз заново вместо того, чтобы подгружать из локального репозитория, либо каждый раз загружать из public-репозитория. Указывать версию как SNAPSHOT нужно в том случае, если проект в работе и всегда нужна самая последняя версия.

Транзитивные зависимости

Начиная со второй версии фреймворка *maven* были введены транзитивные зависимости, которые позволяет избегать необходимости изучения и определения библиотек, которые требуются для самой зависимости. Maven включает их автоматически. В общем случае, все зависимости, используемые в проекте, наследуются от родителей. Ограничений по уровню наследований не существует, что, в свою очередь, может вызвать их сильный рост. В качестве примера можно рассмотреть создание проекта «A», который зависит от проекта «B». Но проект «B», в свою очередь, зависит от проекта «C». Подобная цепочка зависимостей может быть сколь угодно длинной. Как в этом случае поступает *maven* и как связан проект «A» и с проектом «C».

В следующей табличке, позаимствованной с сайта maven, представлен набор правил переноса области *scope*. К примеру, если scope артефакта «B» compile, а он, в свою очередь, подключает библиотеку «C» как provided, то наш проект «A» будет зависеть от «C» так как указано в ячейке находящейся на пересечении строки «compile» и столбца «provided».

	Compile	Provided	Runtime	Test
Compile	Compile	-	Runtime	-
Provided	Provided	Provided	Provided	-
Runtime	Runtime	-	Runtime	-
Test	Test	Test	Test	-

Плагин dependency

Имея приведенную выше таблицу правил переноса scope и набор соответствующих артефактам файлов pom можно построить дерево зависимостей для каждой из фаз жизненного цикла проекта. Строить вручную долго и сложно. Можно использовать maven-плагин *dependency* и выполнить команду «mvn dependency:list», в результате выполнения которой получим итоговый список артефактов и их scope :

```
F:\Projects\example>mvn dependency:list
Scanning for projects...

------------------------------------------------------------------------
Building example 2.1
------------------------------------------------------------------------

--- maven-dependency-plugin:2.8:list (default-cli) @ example ---

The following files have been resolved:
    net.sf.ezmorph:ezmorph:jar:1.0.6:compile
    ru.test:dao:jar:2.11:compile
    net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
    ru.test:iplugin:jar:1.1:compile
    commons-collections:commons-collections:jar:3.2.1:compile
    commons-beanutils:commons-beanutils:jar:1.8.0:compile
    commons-lang:commons-lang:jar:2.5:compile
    org.eclipse.swt.win32.win32.x86:org.eclipse.swt.win32.win32.\
                                    x86:jar:3.7.1.v3738:compile
    commons-logging:commons-logging:jar:1.1.1:compile
```

Однако к такому списку могут возникнуть вопросы : откуда взялся тот или иной артефакт? Т.е. желательно показать транзитные зависимости. И вот, команда «mvn dependency:tree» позволяет сформировать такое дерево зависимостей :

```
F:\Projects\example>mvn dependency:tree
Scanning for projects...

------------------------------------------------------------------------
Building example 2.1
------------------------------------------------------------------------

--- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
 ru.test:example:jar:2.1
  +- net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
  |  +- commons-beanutils:commons-beanutils:jar:1.8.0:compile
  |  +- commons-collections:commons-collections:jar:3.2.1:compile
  |  +- commons-lang:commons-lang:jar:2.5:compile
  |  +- commons-logging:commons-logging:jar:1.1.1:compile
  |  \- net.sf.ezmorph:ezmorph:jar:1.0.6:compile
  +- ru.test:iplugin:jar:1.1:compile
  |  \- ru.test:dao:jar:2.11:compile
  \- org.eclipse.swt.win32.win32.x86:org.eclipse.swt.win32.win32.\
                                     x86:jar:3.7.1.v3738:compile
```

Плагин *dependency* содержит большое количество целей *goal*, к наиболее полезным из которых относятся :

- **dependency:list** – выводит список зависимостей и области их действия scope;
- **dependency:tree** – выводит иерархический список зависимостей и области их действия scope;
- **dependency:purge-local-repository** – служит для удаления из локального репозитория всех артефактов, от которых прямо или косвенно зависит проект. После этого удаленные артефакты загружаются из Internet заново. Это может быть полезно в том случае, когда какой-либо артефакт был загружен со сбоями. В этом случае проще очистить локальный репозиторий и попробовать загрузить библиотеки заново;
- **dependency:sources** - служит для загрузки из центральных репозиториев исходников для всех артефактов, используемых в проекте. Порой отлаживая код, часто возникает необходимость подсмотреть исходный код какой-либо библиотеки;
- **dependency:copy-dependencies** - копирует зависимости/артефакты в поддиректорию target/dependency;
- **dependency:get** - копирует зависимость в локальный репозиторий.

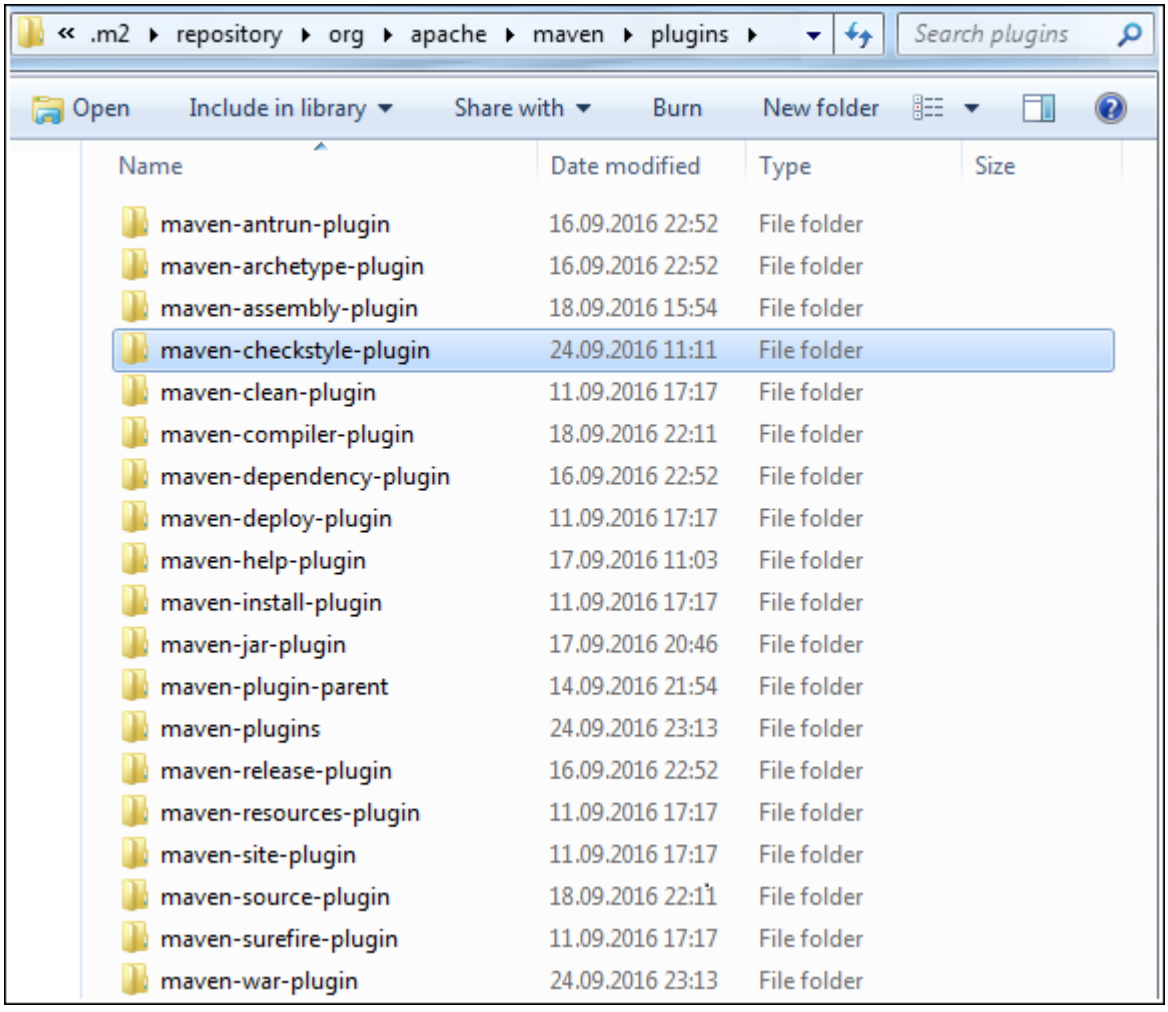
Копирование зависимости в локальный репозиторий

Следующий команда загрузит библиотеку JFreeChart (версия 1.0.19) в локальный репозиторий.

```
mvn dependency:get -Dartifact=org.jfree:jfreechart:1.0.19:jar
```

Maven плагины для сборки проекта

С описанием фреймворка **maven** можно познакомиться здесь. На этой странице рассматриваются наиболее распространенные плагины, используемые при сборке проекта. Список установленных на компьютере плагинов *maven* можно увидеть в директории ${M2_HOME}/repository/org/apache/maven/plugins приблизительно в таком виде, как на следующем скриншоте.



На странице рассмотрены следующие плагины с примерами :

- maven-compiler-plugin - плагин компиляции;
- maven-resources-plugin - плагин включения ресурсов;
- maven-source-plugin - плагин включения исходных кодов;
- maven-dependency-plugin - плагин копирования зависимостей;
- maven-jar-plugin - плагин создания jar-файла;

- maven-surefire-plugin - плагин запуска тестов;
- buildnumber-maven-plugin - плагин генерации номера сборки;

Плагин создания проекта maven-archetype-plugin

Одним из самых первых плагинов, с которым приходится знакомиться или начинать новый проект, это *maven-archetype-plugin*. Данный плагин позволяет по определенному шаблону (archetype) сформировать структуру проекта. Примеры **maven** проектов для разнотипных приложений можно увидеть здесь.

Плагин компиляции maven-compiler-plugin

Самый популярный плагин, позволяющий управлять версией компилятора и используемый практически во всех проектах, это компилятор *maven-compiler-plugin*. Он доступен по умолчанию, но практически в каждом проекте его приходится переобъявлять. В простейшем случае плагин позволяет определить версию java машины (JVM), для которой написан код приложения, и версию java для компиляции кода. Пример использования :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <encoding>UTF-8</encoding>
    </configuration>
</plugin>
```

В данном примере определена версия java-кода 1.7, на котором написана программа (source). Версия java машины, на которой будет работать программа, определена тегом <target>. В теге <encoding> указана кодировка исходного кода (UTF-8). По умолчанию используется версия java 1.3, а кодировка выбирается из операционной системы. Плагин позволяет указать путь к компилятору javac тегом <executable>.

Плагин *maven-compiler-plugin* имеет две цели :

- compiler:compile - компиляция исходников, по умолчанию связана с фазой compile;
- compiler:testCompile - компиляция тестов, по умолчанию связана с фазой test-compile.

Кроме приведёных настроек компилятор позволяет определить аргументы компиляции :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgs>
            <arg>-verbose</arg>
            <arg>-Xlint:all,-options,-path<arg>
        </compilerArgs>
    </configuration>
</plugin>
```

Плагин позволяет даже выполнить компилирование не-java компилятором :

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
```

```
        <compilerId>csharp</compilerId>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.codehaus.plexus</groupId>
            <artifactId>plexus-compiler-csharp</artifactId>
            <version>1.6</version>
        </dependency>
    </dependencies>
</plugin>
```

Плагин копирования ресурсов maven-resources-plugin

Перед сборкой проекта необходимо все ресурсы (файлы изображений, файлы .properties) скопировать в директорию *target*. Для этого используется плагин **maven-resources-plugin**. Пример использования плагина :

```
<plugin>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>copy-resources</id>
            <phase>validate</phase>
            <goals>
                <goal>copy-resources</goal>
            </goals>
            <configuration>
                <outputDirectory>
                       ${basedir}/target/resources
                </outputDirectory>
                <resources>
                    <resource>
                        <directory>src/main/resources/props</directory>
                        <filtering>true</filtering>
                        <includes>
                            <include>**/*.properties</include>
                        </includes>
                    </resource>
                    <resource>
                        <directory>src/main/resources/images</directory>
                        <includes>
                            <include>**/*.png</include>
                        </includes>
                    </resource>
                </resources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Тег <outputDirectory> определяет целевую директорию, в которую будет происходить копирование.

В данном примере **maven** должен положить в директорию *target* всё точно также, как и было в проекте. При копировании ресурсов можно использовать дополнительные возможности *maven-resources-plugin*, позволяющие вносить изменения в файлы свойств. Для этого используется тег <filtering>, который при значении *true* предлагает плагину заглянуть во внутрь файла и при наличии определенных значений заменить их переменными *maven'a*. Файлы изображений не фильтруются. Поэтому ресурсы можно разнести по разным тэгам <resource />.

Дополнительно об использовании фильтрации для корректировки значений в файле свойств .properties можно почитать здесь.

Плагин включения исходных кодов maven-source-plugin

Плагин *maven-source-plugin* позволяет включать в сборку проекта исходный код. Данная возможность особенно полезна, если создается многомодульная архитектура проекта, включающая различные файлы .jar, и требуется отладка отдельных частей. Пример использования *maven-source-plugin* :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

В данном примере в релиз проекта будут включены исходные коды программы.

Плагин копирования зависимостей maven-dependency-plugin

Для копирования зависимостей в директорию сборки используют плагин *maven-dependency-plugin*. Пример копирования библиотек в директорию ${project.build.directory}/lib :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.5.1</version>
    <configuration>
        <outputDirectory>
            ${project.build.directory}/lib/
        </outputDirectory>
        <overWriteReleases>false</overWriteReleases>
        <overWriteSnapshots>false</overWriteSnapshots>
        <overWriteIfNewer>true</overWriteIfNewer>
    </configuration>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Назначение опций :

- outputDirectory - определение директории, в которую будут копироваться зависимости;
- overWriteReleases - флаг необходимости перезаписывания зависимостей при создании релиза;
- overWriteSnapshots - флаг необходимости перезаписывания неокончательных зависимостей, в которых присутствует SNAPSHOT;

- overWriteIfNewer - флаг необходимости перезаписывания библиотек с наличием более новых версий.

По умолчанию <overWriteReleases> и <overWriteSnapshots> - false, для <overWriteIfNewer> - true.

В примере определен раздел <execution> с идентификатором *copy-dependencies* - копирование зависимостей. Плагин используется в фазе сборки <package>, цель *copy-dependencies*. В разделе конфигурации *configuration* определен каталог, в который будут копироваться зависимости. Дополнительные параметры говорят о том, что перезаписываем библиотеки с наличием более новых версий, не перезаписываем текущие версии и не перезаписываем зависимости без окончательной версии (SNAPSHOT).

Плагин *maven-dependency-plugin* включает несколько целей, некоторые приведены ниже :

- mvn dependency:analyze - анализ зависимостей (используемые, неиспользуемые, указанные, неуказанные);
- mvn dependency:analyze-duplicate - определение дублирующиеся зависимостей;
- mvn dependency:resolve - разрешение (определение) всех зависимостей;
- mvn dependency:resolve-plugin - разрешение (определение) всех плагинов;
- mvn dependency:tree - вывод на экран дерева зависимостей.

Плагин создания jar-файла maven-jar-plugin

Плагин *maven-jar-plugin* позволяет сформировать манифест, описать дополнительные ресурсы, необходимые для включения в jar-файл, и упаковать проект в jar-архив. Пример проектного файла pom.xml, описывающий настройку данного плагина :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <includes>
            <include>**/properties/*</include>
        </includes>
        <excludes>
            <exclude>**/*.png</exclude>
        </excludes>
        <archive>
           <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
        </archive>
    </configuration>
</plugin>
```

В примере определена директория и манифест, включаемые в сборку. Тегом <excludes> блокируется включение в сборку определенных файлов изображений.

Плагин *maven-jar-plugin* может создать и включить в сборку MANIFEST.MF самостоятельно. Для этого следует в секцию <archive> включить тег <manifest> с опциями :

```
<configuration>
    <archive>
        <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>lib/</classpathPrefix>
            <mainClass>ru.company.AppMain</mainClass>
        </manifest>
    </archive>
</configuration>
```

Опции тега <manifest> :

- <addClasspath> определяет необходимость добавления в манифест CLASSPATH;
- <classpathPrefix> позволяет дописывать префикс (в примере lib) перед каждым ресурсом;
- <mainClass> указывает на главный исполняемый класс.

Определение префикса в <classpathPrefix> позволяет размещать зависимости в отдельной папке.

Пример создания сборки (исполняемый jar-файл) с зависимостями библиотеки SWT можно посмотреть здесь.

Плагин тестирования maven-surefire-plugin

Плагин *maven-surefire-plugin* предназначен для запуска тестов и генерации отчетов по результатам их выполнения. По умолчанию на тестирование запускаются все java-файлы, наименование которых начинается с «Test» и заканчивается «Test» или «TestCase» :

- **/Test*.java
- **/*Test.java
- **/*TestCase.java

Если необходимо запустить java-файл с отличным от соглашения наименованием, например Sample.java, то необходимо в проектный файл pom.xml включить соответствующую секцию с плагином *maven-surefire-plugin*.

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.12.4</version>
        <configuration>
            <includes>
                <include>Sample.java</include>
            </includes>
        </configuration>
    </plugin>
</plugins>
```

Плагин *maven-surefire-plugin* содержит единственную цель *surefire:test*. Для разработки кодов тестирования можно использовать как JUnit, так и TestNG. Результаты тестирования в виде отчетов в форматах .txt и .xml сохраняются в директории ${basedir}/target/surefire-reports.

Иногда приходится отдельные тесты исключать. Это можно сделать включением в секцию <configuration> тега <excludes>.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.12.4</version>
    <configuration>
        <excludes>
            <exclude>**/TestCircle.java</exclude>
            <exclude>**/TestSquare.java</exclude>
        </excludes>
    </configuration>
</plugin>
```

Чтобы запустить проект на тестирование необходимо выполнить одну из следующих команд :

```
mvn test
```

```
mvn -Dmaven.surefire.debug test
```

Чтобы пропустить выполнение тестов на этапе сборки проекта, можно выполнить команду.

```
mvn clean package -Dmaven.test.skip=true
```

Также можно проигнорировать выполнение тестирования проекта включением в секцию <configuration> тега <skipTests>

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

Плагин генерации номера сборки buildnumber-maven-plugin

Предположим, что нам нужно в манифест MANIFEST.MF нашего WEB-приложения и в файл свойств src/main/resources/app.properties положить номер сборки, который определяется переменной *${buildNumber}*. Файл манифеста будем генерить автоматически. А в файл свойств проекта app.properties включим параметры, значения которых будут определяться на этапе сборки проекта :

```
# application.properties
app.name=${pom.name}
app.version=${pom.version}
app.build=${buildNumber}
```

В режиме выполнения программы (runtime) можно обращаться к данному файлу свойств для получения наименования приложения и номера версии сборки.

Для генерирования уникального номера сборки проекта используется плагин *buildnumber-maven-plugin*. Найти плагин можно в репозитории в директории ${M2_HOME}/repository/org/codehaus/mojo. Пример настройки плагина в проектном файле pom.xml :

```
<modelVersion>4.0.0</modelVersion>
<groupId>ru.hellogwt.sample</groupId>
<artifactId>hellogwt</artifactId>
<packaging>war</packaging>
<version>1.0</version>
<name>GWT Maven Archetype</name>
. . . .
<plugins>
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>buildnumber-maven-plugin</artifactId>
        <version>1.2</version>
        <executions>
            <execution>
                <phase>validate</phase>
                <goals>
                    <goal>create</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <revisionOnScmFailure>true</revisionOnScmFailure>
            <format>{0}-{1,date,yyyyMMdd}</format>
```

```
                <items>
                  <item>${project.version}</item>
                  <item>timestamp</item>
                </items>
            </configuration>
        </plugin>
</plugins>
```

В приведенном примере плагин запускается во время фазы жизненного цикла *validate* и генерирует номер версии ${buildNumber}, который собирается из нескольких частей и определен тэгом <format />. Каждая часть номера версии заключается в фигурные скобки и формируется согласно описанию *MessageFormat* языка Java. Каждой части соответствует тэг <item />, указывающий, какое значение должно быть подставлено.

Если в pom.xml не настроена работа с SCM (Source Code Management) типа Subversion, Git и т.п., то при попытке генерации номера сборки будет получено сообщение об ошибке "The scm url cannot be null". В этом случае можно указать в pom.xml заглушку SCM.

```
<scm>
    <connection>scm:svn:http://127.0.0.1/dummy</connection>
    <developerConnection>scm:svn:https://127.0.0.1/dummy</developerConnection>
    <tag>HEAD</tag>
    <url>http://127.0.0.1/dummy</url>
</scm>
```

Чтобы номер сборки генерировался независимо от подключения к SCM в настройках конфигурации плагина следует указать свойство *revisionOnScmFailure* равным true.

Теперь настроим плагин *maven-war-plugin*, чтобы номер версии поместить в MANIFEST.MF, который будет создаваться автоматически :

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
            <manifest>
                <addDefaultImplementationEntries>
                    true
                </addDefaultImplementationEntries>
            </manifest>
            <manifestEntries>
                <Implementation-Build>
                    ${buildNumber}
                </Implementation-Build>
            </manifestEntries>
        </archive>
    </configuration>
</plugin>
```

Генерируемое значение, по-умолчанию, сохраняется в файле ${basedir}/buildNumber.properties и имеет имя buildNumber. При необходимости данные параметры могут быть переопределены через свойства buildNumberPropertiesFileLocation и buildNumberPropertyName соответственно.

Чтобы определить значения в файле свойств src/main/resources/app.properties включим фильтрацию ресурсов в разделе <build> :

```
<build>
    <resources>
```

```
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
     ....
</build>
```

Дополнительную информацию о фильтрации для корректировки значений в файле свойств *.properties можно почитать здесь.

После выполнения сборки проекта манифест MANIFEST.MF в файле .war будет иметь приблизительно следующий вид :

```
Manifest-Version: 1.0
Implementation-Title: GWT Maven Archetype
Implementation-Version: 1.0
Implementation-Vendor-Id: ru.hellogwt.sample
Built-By: Father
Build-Jdk: 1.7.0_67
Created-By: Apache Maven 3.3.9
Implementation-Build: 1.0-20161001
Archiver-Version: Plexus Archiver
```

Конечно же изменится и файл свойств app.properties в сборке :

```
# application.properties
app.name=GWT Maven Archetype
app.version=1.0
app.build=1.0-20161001
```

Чтобы в сервлете WEB-приложения прочитать версию сборки в файле MANIFEST.MF можно использовать следующий код :

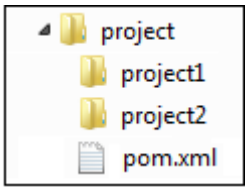
```
import java.io.IOException;
import java.util.Properties;
...
String version = "UNDEFINED";
Properties prop = new Properties();
try {
    prop.load(getServletContext().getResourceAsStream("/META-INF/MANIFEST.MF"));
    version = prop.getProperty("Implementation-Build");
} catch (IOException e) {}
...
```

Наследование проектов в maven

Одним из важных аспектов многомодульного приложения является возможность независимой разработки отдельных модулей, обеспечивая, таким образом, расширение и изменение функциональности системы в целом. И здесь существенную помощь разработчикам оказывает фреймворк maven, который позволяет связать все проекты системы в единое целое. Чтобы объединить несколько maven-проектов в один связанный проект необходимо использовать **наследование**, которое определяет включение дополнительных секций в pom.xml (POM - Project Object Model).

Допустим необходимо разработать два взаимосвязанных проекта (project1, project2), которые должны быть объединены в едином родительском проекте *project*. Физически проекты необходимо расположить в одной родительской директории, в которой дочерние maven-

проекты являются поддиректориями. Родительский файл pom.xml располагается в корневой директории, как это представлено на следующем скриншоте :



Настройка родительского pom.xml

В pom.xml родительского проекта необходимо определить параметры GAV (groupId, artifactId, version) и в теге <packaging> указать значение «pom». Дополнительно вводится секция <modules>, в которой перечисляются все дочерние проекты.

```
<groupId>com.example</groupId>
<artifactId>project</artifactId>
<version>0.0.1</version>
<packaging>pom</packaging>

. . .

<modules>
    <module>project1</module>
    <module>project2</module>
</modules>
```

Настройка дочерних pom.xml

В pom.xml дочерних проектов необходимо ввести секцию <parent> и определить GAV-параметры родительского проекта.

```
<parent>
    <groupId>com.example</groupId>
    <artifactId>project</artifactId>
    <version>0.0.1</version>
</parent>
```

На этом можно сказать, что все дочерние проекты привязаны к родительскому.

Применение наследования maven

Наследование в maven-проектах широко используется при разработке плагинов/бандлов для контейнеров OSGi (Open Services Gateway Initiative) и компонентов EJB (Enterprise JavaBeans). Также можно использовать «преимущества» наследования и в простых проектах.

Зачем объединять проекты?

В связанных многомодульных проектах можно исключить дублирование свойств и зависимостей, а также использовать централизованное управление и контролировать зависимости.

1. Общие свойства проектов

Связанные проекты позволяют определить общие свойства проектов, зависимости и разместить их в родительском pom.xml. Дочерние проекты будут автоматически наследовать свойства родителя. В следующем примере создаются общие секции <properties> и <dependencies>. В секцию зависимостей включены junit и log4j.

```
<properties>
    <junit.version>4.11</junit.version>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>
```

Дочерние объекты наследуют значения следующих GAV-параметров родителя : <groupId>, <version>, но их можно при необходимости переопределить.

2. Централизованное управление проектами

Выполняя maven-команды в родительском проекте, они автоматически будут выполнены для каждого из подпроектов. В следующем примере выполняется команда install, согласно которой после сборки всех проектов они будут размещены в локальном репозитории.

```
$ cd project

$ mvn install
```

Таким образом, можно выполнять *maven* команды как для отдельного подпроекта, перемещаясь в его поддиректорию, так и для всех проектов вместе, располагаясь в родительской директории.

Чтобы посмотреть список зависимостей проекта/ов, необходимо выполнить команду «mvn dependency:tree» :

```
$ cd project

$ mvn dependency:tree

[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Main project
[INFO] Chaild project1
[INFO] Chaild project2
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Main project 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project ---
[INFO] com.example:project:pom:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
```

```
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Chaild project1 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project1 ---
[INFO] com.example:project1:jar:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Chaild project2 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project2 ---
[INFO] com.example:project2:jar:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO] ------------------------------------------------------------------------
```

Maven сначала выводит в консоль зависимости главного проекта, а потом зависимости для каждого из дочерних проектов. Как видно в примере значения groupId (com.example) и version (0.0.1) дочерних проектов совпадают с родительским. Они теперь необязательны и берутся по умолчанию у parent проекта, хотя можно определить собственные значения для каждого подпроекта.

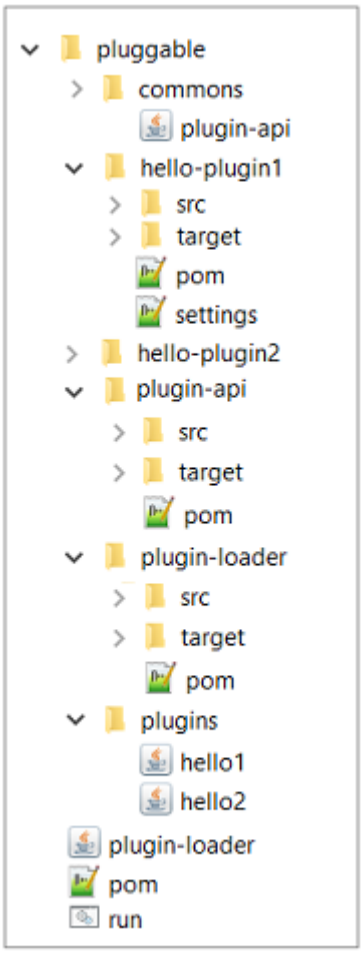
Кроме свойств <properties> и зависимостей <dependencies> в родительском проекте часто объявляют необходимые для сборки плагины и репозитории.

Многомодульный maven проект

Maven позволяет собирать проект из нескольких модулей. Каждый программный модуль включает свой проектный файл pom.xml. Один из проектных pom.xml файлов является корневым. Корневой pom.xml позволяет объединить все модули в единый проект. При этом в корневой проектный файл можно вынести общие для всех модулей свойства. А каждый модульный pom.xml должен включать параметры GAV (groupId, artifactId, version) корневого pom.xml.

Общие положения разработки многомодульного maven-приложения рассмотрены на странице Наследование проектов в maven. В данной статье рассмотрим пример сборки многомодульного приложения. В качестве «подопытного» приложения используем пример, представленный на странице Pluggable решение. На выходе данного примера мы должны получить 3 архивных и один исполняемый jar-файлов. Главный исполняемый jar-модуль динамически «при необходимости» загружает остальные архивные jar'ники. Данный «подопытный» пример был использован для «оборачивания» jar'ника в exe-файл с использованием maven-плагина launch4j.

Описание многомодульного проекта



На скриншоте представлена структура проекта pluggable, включающая следующие проектные модули :

- hello-plugin1 – динамически загружаемый плагин №1 (hello1.jar);
- hello-plugin2 – динамически загружаемый плагин №2 (hello2.jar);
- plugin-api – интерфейсы описания плагинов (plugin-api.jar);
- plugin-loader – главный исполняемый jar модуль.

Дополнительные поддиректории проекта, используемые для размещения jar-модулей :

- commons – поддиректория размещения архивного jar-модуля описания интерфейса плагинов;
- plugins – поддиректория размещения jar-модулей (плагинов);

Главный исполняемый модуль plugin-loader.jar размещается в корневой директории проекта, где размещается и проектный/корневой pom.xml. Файл run.bat можно использовать для старта plugin-loader.jar из консоли в Windows.

Примечание : в исходные коды классов внесены изменения, связанные с из размещением в пакетах. В исходном примере все классы располагаются в «корне».

Начнем рассмотрение примера с корневого многомодульного pom.xml.

Листинг многомодульного корневого pom.xml

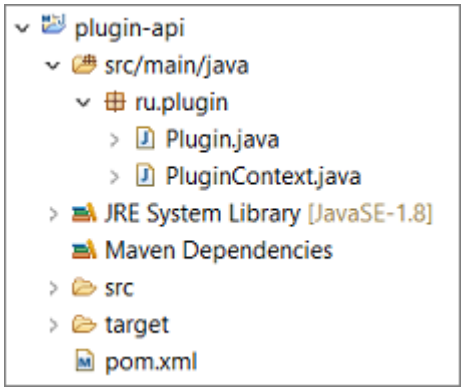
Корневой pom.xml включает параметры GAV (groupId, artifactId, Version), общую для всех модулей проекта секцию <properties> и секцию описания модулей <modules>. Обратите внимание на атрибут <**packaging**>, значение которого должно быть «**pom**».

Следует отметить, что порядок включения программных модулей проекта составлен таким образом, что сначала представлены исполняемый модуль plugin-loader.jar и плагины (hello-plugin1.jar, hello-plugin2.jar), после чего следует интерфейсный модуль plugin-api.jar. Если собирать проект по-отдельности, то модуль plugin-api.jar должен быть собран в первую очередь и размещен в репозитории командой «mvn install». В этом случае зависимые модули plugin-loader.jar и плагины (hello-plugin1, hello-plugin2) собрались бы нормально. Ну, а мы в этом примере посмотрим, как поступит Maven в случае, если порядок описания модулей для сборки «нарушен».

```
<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>ru.pluggable.main</groupId>
    <artifactId>pluggable</artifactId>
    <packaging>pom</packaging>
    <version>1.0.0</version>
    <name>MultiModule application</name>

    <properties>
        <maven.test.skip>true</maven.test.skip>
        <java.version>1.8</java.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>
            UTF-8
        </project.build.sourceEncoding>
    </properties>

    <modules>
        <module>plugin-loader</module>
        <module>hello-plugin1</module>
        <module>hello-plugin2</module>
        <module>plugin</module>
    </modules>
</project>
```

Модуль описания интерфейсов плагинов plugin-api.jar

На следующем скриншоте представлена структура проекта plugin-api. Интерфейсные классы Plugin, PluginContext располагаются в пакете «ru.plugin».



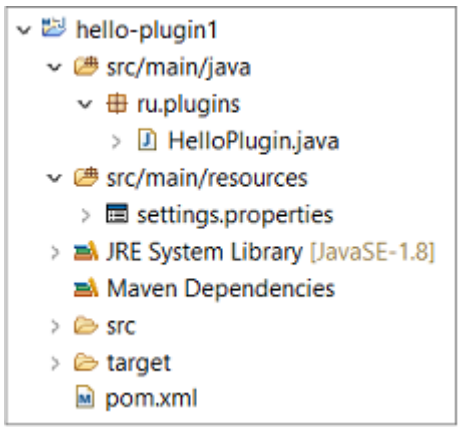
Листинг pom.xml

Проектный pom.xml модуля plugin-api.jar включает GAV-параметры, секцию описания родительского GAV (<parent>) и секцию сборки <build>, где в качестве выходной директории размещения (<outputDirectory>) указана поддиректория «${basedir}/../commons».

```
<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>ru.pluggable</groupId>
    <artifactId>plugin-api</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0</version>
    <name>Plugin API</name>

    <parent>
        <groupId>ru.pluggable.main</groupId>
        <artifactId>pluggable</artifactId>
        <version>1.0.0</version>
    </parent>

    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>
                    org.apache.maven.plugins
                </groupId>
                <artifactId>
                    maven-jar-plugin
                </artifactId>
                <version>2.3.1</version>
                <configuration>
                    <outputDirectory>
                        ${basedir}/../commons
                    </outputDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Модуль описания плагина hello-plugin1.jar

Структура проекта hello-plugin1 представлена на следующем скриншоте.



Класс HelloPlugin, расположенный в пакете «ru.plugins», реализует свойства интерфейса Plugin. При инициализации класса в методе init определяется значение контекста PluginContext родительского/вызвавшего объекта. Метод invoke выводит в консоль сообщение и изменяет надпись на кнопке родительского объекта.

```
package ru.plugins;

import ru.plugin.Plugin;
import ru.plugin.PluginContext;

public class HelloPlugin implements Plugin
{
    private PluginContext pc;

    @Override
    public void invoke() {
        System.out.println("Hello world. I am a plugin 1");
        pc.getButton().setText("Other text 1");
    }
    @Override
    public void init(PluginContext context) {
        this.pc = context;
    }
}
```

Листинг pom.xml

Проектный pom.xml модуля hello-plugin1.jar включает GAV-параметры, секцию описания родительского GAV (<parent>), секцию зависимостей (<dependencies>) и секцию сборки <build>. В секции зависимостей указываются параметры модуля plugin-api. В описании секции сборки используется 2 плагина (maven-resources-plugin, maven-jar-plugin). Первый плагин включает в сборку ресурсы (resources/settings.properties), второй плагин создает jar и размещает его в выходной директории (<outputDirectory>) «${basedir}/../plugins».

```
<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>ru.plugins</groupId>
    <artifactId>hello1</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0</version>
    <name>Plugin Hello1</name>

    <parent>
        <groupId>ru.pluggable.main</groupId>
        <artifactId>pluggable</artifactId>
        <version>1.0.0</version>
```

```
    </parent>

    <dependencies>
       <dependency>
            <groupId>ru.pluggable</groupId>
            <artifactId>plugin-api</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>

    <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <artifactId>
                maven-resources-plugin
            </artifactId>
            <version>2.6</version>
            <executions>
                <execution>
                    <id>copy-resources</id>
                    <configuration>
                        <outputDirectory>
                            ${basedir}/target/resources
                        </outputDirectory>
                        <resources>
                            <directory>
                                src/main/resources
                            </directory>
                            <resource>
                                <includes>
                                    <include>
                                        **/*.properties
                                    </include>
                                </includes>
                            </resource>
                        </resources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.3.1</version>
            <configuration>
                <outputDirectory>
                    ${basedir}/../plugins
                </outputDirectory>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

Примечание : второй плагин hello-plugin2 структурно ничем не отличается от hello-plugin1. Отличия касаются текста сообщения в консоли, надписи на кнопке и параметров GAV в pom.xml.

Проектный pom.xml модуля plugin-loader

Проектный pom.xml включает GAV-параметры jar-модуля, секцию описания родительского GAV (<parent>), секцию зависимостей (<dependencies>) и секцию сборки <build>. В секции зависимостей указываются параметры модуля plugin-api. В описании секции сборки используется плагин maven-jar-plugin, который создает jar и размещяет его в корневой директории проекта (<outputDirectory>).

```xml
<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>ru.pluggable.loader</groupId>
    <artifactId>plugin-loader</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0</version>
    <name>Plugin Loader</name>

    <parent>
        <groupId>ru.pluggable.main</groupId>
        <artifactId>pluggable</artifactId>
        <version>1.0.0</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>ru.pluggable</groupId>
            <artifactId>plugin-api</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>

    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                    <outputDirectory>
                        ${basedir}/..
                    </outputDirectory>
                    <archive>
                        <manifest>
                            <mainClass>
                                ru.pluggable.loader.Boostrap
                            </mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Сборка проекта

Сборка всех проектов выполняется одной командой «mvn package» для корневого pom.xml. Maven сначала просматривает проектные файлы pom.xml всех модулей, определенных в корневом pom.xml, и после этого определяет порядок сборки модулей. Как следует из представленных ниже сообщений, выводимых Maven в консоль, порядок сборки был изменен и первым собирается модуль Plugin API, после чего формируются зависимые от него Plugin Loader, Plugin Hello1, Plugin Hello2.

```
D:\pluggable>mvn package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] MultiModule application
```

```
[INFO] Plugin API
[INFO] Plugin Loader
[INFO] Plugin Hello1
[INFO] Plugin Hello2
[INFO]
[INFO] ------------------------------------------------
[INFO] Building MultiModule application 1.0.0
[INFO] ------------------------------------------------
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin API 1.0.0
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources \
          (default-resources) @ plugin-api ---
[INFO] Using 'UTF-8' encoding to copy filtered \
          resources.
[INFO] skip non existing resourceDirectory \
          D:\pluggable\plugin\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile \
         (default-compile) @ plugin-api ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to \
          D:\pluggable\plugin\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources \
          (default-testResources) @ plugin-api ---
[INFO] Not copying test resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile \
          (default-testCompile) @ plugin-api ---
[INFO] Not compiling test sources
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test \
          (default-test) @ plugin-api ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) \
          @ plugin-api ---
[INFO] Building jar: \
          D:\pluggable\plugin\..\commons\plugin-api.jar
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Loader 1.0.0
[INFO] ------------------------------------------------
[INFO]
. . .
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Hello1 1.0.0
[INFO] ------------------------------------------------
[INFO]
. . .
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Hello2 1.0.0
[INFO] ------------------------------------------------
. . .
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) \
         @ hello2 ---
[INFO] Building jar: \
         D:\pluggable\hello-plugin2\..\plugins\hello2.jar
[INFO] ------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] MultiModule application ..... SUCCESS [  0.006 s]
[INFO] Plugin API .................. SUCCESS [  1.957 s]
[INFO] Plugin Loader ............... SUCCESS [  0.391 s]
[INFO] Plugin Hello1 ............... SUCCESS [  0.183 s]
```

```
[INFO] Plugin Hello2 ..... SUCCESS [  0.115 s]
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time: 2.802 s
[INFO] Finished at: 2019-06-05T11:07:45+03:00
[INFO] Final Memory: 18M/199M
[INFO] ------------------------------------------------------
```

Примечание :
1. При первой сборке проекта maven не нашел в репозитории модуль plugin-api.jar, прекратил сборку и вывел соответствующее сообщение. После размещения модуля plugin-api.jar в локальном репозитории командой «mvn install» сборка всего проекта прошла успешно.
2. После первой сборки проекта из локального репозитория был удален модуль plugin-api.jar. При дальнейших запусках сборки проекта Maven больше не ругался и сборка проходила нормально. Каким образом Maven собирает проект при отсутствии в локальном репозитории зависимого plugin-api.jar для меня осталось загадкой

Maven репозиторий внутри проекта

При поиске зависимостей **maven** последовательно просматривает ряд репозиториев, начиная с локального, расположенного по умолчанию в директории ${user.home}/.m2/repository. Если зависимость в локальном репозитории отсутствует, то далее *maven* выполняет поиск во внутреннем «корпоративном» репозитории (если он используется), и после этого в центральном репозитории. Как только зависимость найдена, то *maven* закачивает ее в локальный репозиторий.

Задачу *maven'y* можно «облегчить» размещением репозитория внутри проекта. Только после этого необходимо в проектном файле *pom.xml* указать местоположение репозитория.

Создание maven репозитория в проекте

Для создания в проекте maven репозитория необходимо выполнить команду **mvn install:install-file**. Создадим в корне проекта ${basedir} директорию *repo*, которую будем использовать в качестве проектного репозитория. Чтобы *maven* загрузил «зависимость» в *repo*, необходимо использовать опцию *localRepositoryPath*. *Maven команды* для создания репозитория будут выглядеть следующим образом.

```
mvn install:install-file
    -Dfile=org.eclipse.swt.win32.win32.x86_3.7.1.v3738a.jar
    -DgroupId=org.eclipse.swt.win32.win32.x86
    -DartifactId=org.eclipse.swt.win32.win32.x86
    -Dversion=3.7.1.v3738
```

```
    -Dpackaging=jar
    -DlocalRepositoryPath=repo
    -DgeneratePom=true
mvn install:install-file
    -Dfile=org.eclipse.core.runtime_3.7.0.v20110110.jar
    -DgroupId=org.eclipse.core.runtime
    -DartifactId=org.eclipse.core.runtime
    -Dversion=3.7.0.v20110110
    -Dpackaging=jar
    -DlocalRepositoryPath=repo
    -DgeneratePom=true
mvn install:install-file
    -Dfile=org.eclipse.core.commands_3.6.0.I20110111-0800.jar
    -DgroupId=org.eclipse.core.commands
    -DartifactId=org.eclipse.core.commands
    -Dversion=3.6.0.I20110111-0800
    -Dpackaging=jar
    -DlocalRepositoryPath=repo
    -DgeneratePom=true
```

После выполнения данных команд в поддиректории проекта *repo* будет создан проектный репозиторий. Необходимо сказать, что *maven команды* должны быть однострочные. Здесь на сайте каждая опция представлена отдельной строкой только для наглядности.

При выполнении команды «mvn install:install-file» *maven* выведет в консоль информацию приблизительно следующего содержания :

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building swit 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-install-plugin:2.4:install-file (default-cli) @ \
                                                          swit-repo ---
[INFO] Installing E:\swit-repo\
       org.eclipse.core.commands_3.6.0.I20110111-0800.jar to
       E:\swit-repo\repo\org\eclipse\core\commands\
       org.eclipse.core.commands\3.6.0.I20110111-0800\org.eclipse.core.
       commands-3.6.0.I20110111-0800.jar
[INFO] Installing C:\Users\PC\AppData\Local\Temp\
       mvninstall5195949622522809983.pom
       to E:\swit-repo\repo\org\eclipse\core\commands\
       org.eclipse.core.commands\3.6.0.I20110111-0800\org.eclipse.core.
       commands-3.6.0.I20110111-0800.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.953 s
[INFO] Finished at: 2016-10-16T16:33:30+04:00
[INFO] Final Memory: 5M/15M
[INFO] ------------------------------------------------------------------------
```

Подключение репозитория

Для подключения репозитория необходимо в проектном файле *pom.xml* разместить следующий код :

```
<repositories>
    <repository>
        <id>repo</id>
        <releases>
            <enabled>true</enabled>
```

```
            <checksumPolicy>ignore</checksumPolicy>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <url>file://${basedir}/repo</url>
    </repository>
</repositories>
```

Теперь при первой загрузке зависимости из проектного репозитория в локальный *maven* выведет в консоль приблизительно следующую информацию :

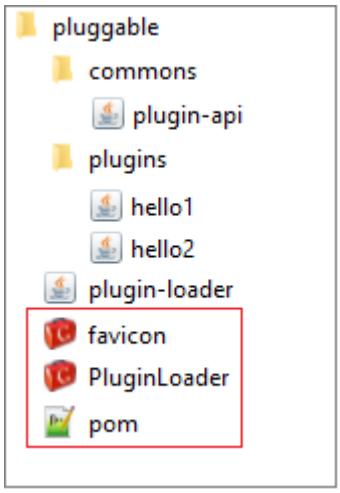
```
E:\swit-repo>mvn clean package -Dmaven.test.skip=true
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building swit 1.0
[INFO] ------------------------------------------------------------------------
Downloading: file://E:/swit-repo/repo/org/eclipse/core/runtime/
                  org.eclipse.core.runtime/3.7.0.v20110110/
                  org.eclipse.core.runtime-3.7.0.v20110110.pom
Downloaded: file://E:/swit-repo/repo/org/eclipse/core/runtime/
                  org.eclipse.core.runtime/3.7.0.v20110110/
                  org.eclipse.core.runtime-3.7.0.v20110110.pom
                  (502 B at 9.8 KB/sec)
Downloading: file://E:/swit-repo/repo/org/eclipse/core/commands/
                  org.eclipse.core.commands/3.6.0.I20110111-0800/
                  org.eclipse.core.commands-3.6.0.I20110111-0800.pom
Downloaded: file://E:/swit-repo/repo/org/eclipse/core/commands/
                  org.eclipse.core.commands/3.6.0.I20110111-0800/
                  org.eclipse.core.commands-3.6.0.I20110111-0800.pom
                  (509 B at 16.6 KB/sec)
Downloading: file://E:/swit-repo/repo/org/eclipse/equinox/common/
                  org.eclipse.equinox.common/3.6.0.v20110523/
                  org.eclipse.equinox.common-3.6.0.v20110523.pom
Downloaded: file://E:/swit-repo/repo/org/eclipse/equinox/common/
                  org.eclipse.equinox.common/3.6.0.v20110523/
                  org.eclipse.equinox.common-3.6.0.v20110523.pom
                  (506 B at 24.7 KB/sec)
 . .
```

Создание exe-файла из jar

Пользователям Windows привычнее использовать исполняемое приложение в виде exe-файла, нежели архивного jar-файла. Разработчики настольных java-приложений могут плагином **launch4j** не только обернуть исполняемый архивный jar-файл в оболочку exe-файла, но и включить в него иконку, автора, версию. Также данный плагин позволяет определить минимальную версию используемой JRE. В данной статье рассмотрим использование maven-плагина **launch4j** для получения exe-файла.

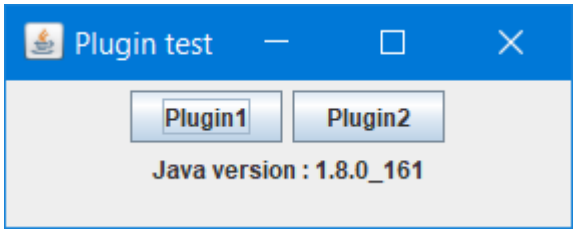
Описание java-примера

В качестве java-примера используем pluggable решение, включающее несколько jar-файлов. На следующем скриншоте представлена структура нашего экспериментального примера. Три файла, выделенные красным прямоугольником и относящиеся к задаче создания исполняемого exe-файла, рассматриваются ниже.



Несколько слов о структуре примера. Описание с исходными кодами данного java-примера представлено на странице Pluggable решение. Желающие могут поближе познакомиться с технологией динамической загрузки jar-файлов (классов), открыв страницу с подробным описанием исходников. На «выходе» данного примера получаем главный исполняемый модуль *plugin-loader.jar*, который использует *common/plugin-api.jar* для загрузки при необходимости (вызове) плагинов *plugins/hello1.jar* и *plugins/hello2.jar*.

Графический интерфейс примера, представленный на следующем скриншоте, включает 2 кнопки с надписями 'Plugin1' и 'Plugin2'. При нажатии на одну из кнопок приложение подгружает необходимый плагин, который меняет надпись на кнопке.



Сообщения в консоли

Динамически загружаемые плагины выводят в консоль дополнительно сообщения. Ниже представлены сообщения от двух плагинов.

```
Hello world. I am a plugin 1
I am a plugin 2
```

Изменения в исходных кодах

Необходимо отметить, что в модули PluginLoader.java и Boostrap.java были внесены изменения. Так в PluginLoader.java добавлена метка JLabel с отображением в интерфейсе версии Java :

```
    . . .
    JLabel label = new JLabel("Java version : " +
System.getProperty("java.version"));
    label.setSize(200, 24);
    frame.getContentPane().add(label);
```

```
    . . .
```

В класс Boostrap.java внесены изменения, связанные с чтением классов (*.class) из jar'ника, а не из директории bin, как это представлено в исходных кодах. Если этого не сделать, то придётся с собой ещё «таскать» и директорию bin с class'ами.

Листинг класса Boostrap.java

В главный класс Boostrap внесены изменения определения url : ниже исходной закомментированной строки размещается код определения url в jar-файле.

```
import java.io.File;
import java.lang.reflect.Method;

import java.net.URL;
import java.net.URLClassLoader;

public class Boostrap {

    public static void main(String[] args) throws Exception
    {
        File commonsDir = new File("commons");

        File[] entries = commonsDir.listFiles();
        URL[] urls = new URL[entries.length];

        for (int i = 0; i < entries.length; i++)
            urls[i] = entries[i].toURI().toURL();

        URLClassLoader loader;
        loader = new URLClassLoader(urls, null);

//      URL url = new File("bin").toURI().toURL();

        File file = new File(".");
        String path = "jar:file:/" + file.getCanonicalPath();
        URL url = new URL(path+"/plugin-loader.jar!/");

        URLClassLoader appLoader;
        appLoader = new URLClassLoader(new URL[]{url},loader);

        Class<?> appClass = loader.loadClass("PluginLoader");
        Object appInstance = appClass.newInstance();
        Method m = appClass.getMethod("start");
        m.invoke(appInstance);
    }
}
```

Оборачивание исполняемого jar в exe-файл

Обычно плагин **maven.plugins.launch4j** включают в проектный pom.xml файл, в котором формируется и исполняемый jar-файл. Поскольку основная цель данной статьи наглядно продемонстрировать возможность оборачивания jar в exe, то уберем из проектного pom.xml все лишнее, что связано с формированием jar-файла. Правильнее сказать создадим такой pom.xml, который и будет решать основную задачу оборачивания jar в exe.

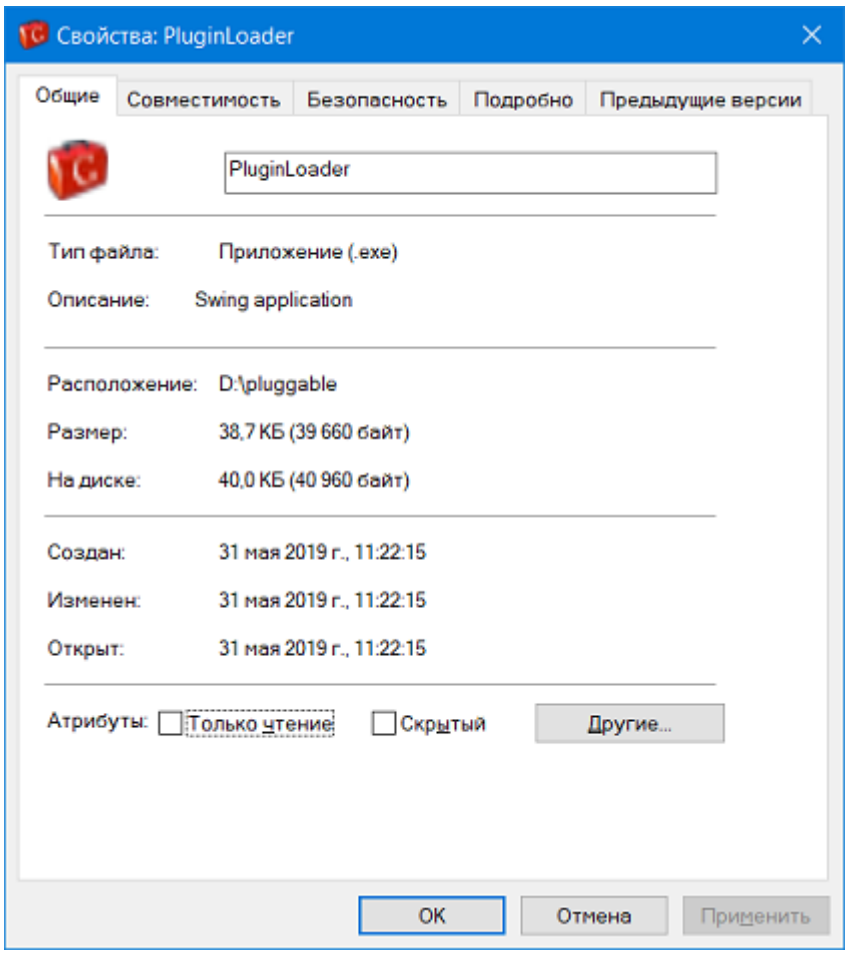
Следующий листинг проектного файла pom.xml решает данную задачу. Сам pom.xml существенно упростился и стал более наглядным. В разделе <properties> определяются наименование компании (product.company) и наименование исполняемого файла (exeFileName), а также минимальная версия jdkVersion. Основные настройки плагина определяются в разделе <executions>. В секции <configuration> указываются jar-файл, exe-файл (outfile) и иконка испольняемого файла (icon). Плагин будет ругаться, если не укажете

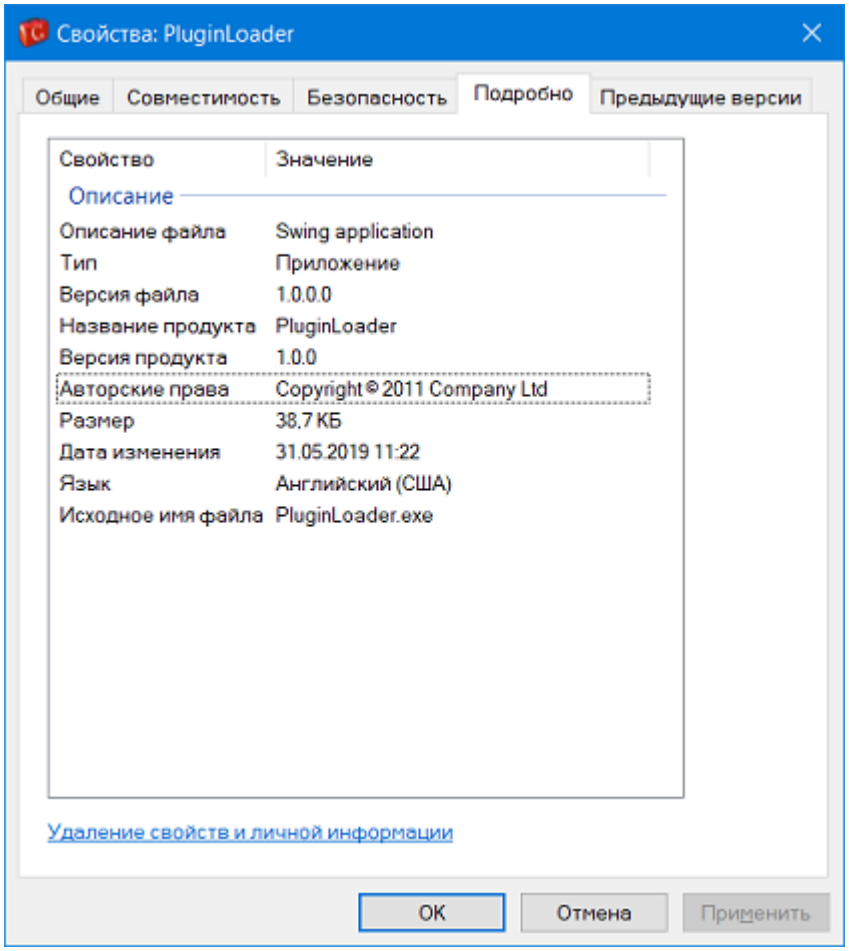
наименование иконки. Следует отметить, что в секции <classPath> необходимо указать главный стартуемый java-класс (mainClas).

```
<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>com.demo</groupId>
  <artifactId>plugin-loader</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>plugin-loader</name>

  <properties>
      <jdkVersion>1.8</jdkVersion>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>

      <project.build.sourceEncoding>
          UTF-8
      </project.build.sourceEncoding>
      <product.company>MultiModule</product.company>
      <product.title>PluginLoader</product.title>
      <exeFileName>PluginLoader</exeFileName>
  </properties>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
    <plugin>
        <groupId>
            com.akathist.maven.plugins.launch4j
        </groupId>
        <artifactId>launch4j-maven-plugin</artifactId>
        <executions>
            <execution>
                <id>plugin-loader</id>
                <phase>package</phase>
                <goals>
                    <goal>launch4j</goal>
                </goals>
                <configuration>
                    <headerType>gui</headerType>
                    <outfile>${exeFileName}.exe</outfile>
                    <jar>${project.artifactId}.jar</jar>
                    <errTitle>${product.title}</errTitle>
                    <icon>favicon.ico</icon>
                    <classPath>
                        <mainClass>Boostrap</mainClass>
                        <addDependencies>
                            true
                        </addDependencies>
                        <preCp>anything</preCp>
                    </classPath>
                    <jre>
                        <minVersion>
                           ${jdkVersion}
                        </minVersion>
                    </jre>
                    <versionInfo>
                        <fileVersion>
                            ${project.version}
                        </fileVersion>
                        <txtFileVersion>
                            ${project.version}
                        </txtFileVersion>
                        <fileDescription>
```

```
                        Swing application
                    </fileDescription>
                    <copyright>
                      Copyright © 2011 ${product.company}
                    </copyright>
                    <productVersion>
                        ${project.version}
                    </productVersion>
                    <txtProductVersion>
                        ${project.version}
                    </txtProductVersion>
                    <companyName>
                        ${product.company}
                    </companyName>
                    <productName>
                        ${product.title}
                    </productName>
                    <internalName>
                        ${exeFileName}
                    </internalName>
                    <originalFilename>
                        ${exeFileName}.exe
                    </originalFilename>
                </versionInfo>
            </configuration>
        </execution>
    </executions>
  </plugin>
  </plugins>
 </build>
</project>
```

На следующих скриншотах представлены вкладки свойств созданного PluginLoader.exe.




Свойства: PluginLoader
Общие
Совместимость
Безопасность
Подробно
Предыдущие версии
Свойство
Значение
Описание
Описание файла
Swing application
Тип
Приложение
Версия файла
1.0.0.0
Название продукта
PluginLoader
Версия продукта
1.0.0
Авторские права
Copyright © 2011 Company Ltd
Размер
38,7 КБ
Дата изменения
31.05.2019 11:22
Язык
Английский (США)
Исходное имя файла
PluginLoader.exe
Удаление свойств и личной информации
OK
Отмена
Применить

Gradle — Обзор

«Gradle — система автоматизации сборки с открытым исходным кодом»

Ant и Maven достигли значительных успехов на рынке JAVA. Ant был первым инструментом сборки, выпущенным в 2000 году, и он разработан на основе идеи процедурного программирования. Позже он улучшен благодаря возможности принимать плагины и управление зависимостями по сети с помощью Apache-IVY. Основным недостатком является XML как формат для написания сценариев сборки, поскольку иерархическая структура не подходит для процедурного программирования, а XML имеет тенденцию становиться неуправляемо большим.

Maven представлен в 2004 году. Он значительно улучшен по сравнению с ANT. Он меняет свою структуру и продолжает использовать XML для написания спецификаций сборки. Maven опирается на соглашения и может загружать зависимости по сети. Основным преимуществом maven является его жизненный цикл. Следуя одному и тому же жизненному циклу для нескольких проектов непрерывно. Это связано с гибкостью. Maven также сталкивается с некоторыми проблемами в управлении зависимостями. Он плохо обрабатывает конфликты между версиями одной и той же библиотеки, и сложные настраиваемые сценарии сборки на самом деле труднее писать в maven, чем в ANT.

Наконец, в 2012 году появился Gradle. Gradle обладает некоторыми эффективными функциями обоих инструментов.

Особенности Gradle

Ниже приведен список функций, которые предоставляет Gradle.

- **Декларативные сборки и сборка по соглашению** — Gradle доступен с отдельным предметно-ориентированным языком (DSL) на основе языка Groovy. Gradle предоставляет элементы декларативного языка. Эти элементы также обеспечивают поддержку сборки по соглашению для Java, Groovy, OSGI, Web и Scala.

- **Язык программирования на основе зависимостей** . Декларативный язык находится на вершине графа задач общего назначения, который вы можете полностью использовать в своей сборке.

- **Структурируйте свою сборку** — Gradle наконец-то позволяет вам применять общие принципы проектирования к вашей сборке. Это даст вам идеальную структуру для сборки, так что вы сможете разработать хорошо структурированную и легко поддерживаемую, понятную сборку.

- **Глубокий API** — Используя этот API, он позволяет вам отслеживать и настраивать его конфигурацию и поведение при выполнении.

- **Масштабирование Gradle** — Gradle может легко увеличить свою производительность, от простых сборок одного проекта до огромных многопроектных сборок предприятия.

- **Многопроектные сборки** — Gradle поддерживает многопроектные сборки и поддерживает частичные сборки. Если вы строите подпроект, Gradle позаботится о создании всех подпроектов, от которых он зависит.

- **Различные способы управления вашими сборками** — Gradle поддерживает различные стратегии для управления вашими зависимостями.

- **Gradle — это первый инструмент интеграции сборки** — Gradle полностью поддерживается для задач ANT, инфраструктура репозитория Maven и lvy для публикации и получения зависимостей. Gradle также предоставляет конвертер для превращения Maven pom.xml в скрипт Gradle.

- **Легкость миграции** — Gradle может легко адаптироваться к любой вашей структуре. Поэтому вы всегда можете разработать свою сборку Gradle в той же ветке, где вы можете создать живой скрипт.

- **Gradle Wrapper** — Gradle Wrapper позволяет выполнять сборки Gradle на машинах, где Gradle не установлен. Это полезно для непрерывной интеграции серверов.

- **Бесплатный открытый исходный код** — Gradle — это проект с открытым исходным кодом, который распространяется под лицензией Apache Software License (ASL).

- **Groovy** — скрипт сборки Gradle написан на Groovy. Весь дизайн Gradle ориентирован на использование в качестве языка, а не жесткой структуры. А Groovy позволяет вам написать собственный скрипт с некоторыми абстракциями. Весь API Gradle полностью разработан на языке Groovy.

Декларативные сборки и сборка по соглашению — Gradle доступен с отдельным предметно-ориентированным языком (DSL) на основе языка Groovy. Gradle предоставляет элементы декларативного языка. Эти элементы также обеспечивают поддержку сборки по соглашению для Java, Groovy, OSGI, Web и Scala.

Язык программирования на основе зависимостей . Декларативный язык находится на вершине графа задач общего назначения, который вы можете полностью использовать в своей сборке.

Структурируйте свою сборку — Gradle наконец-то позволяет вам применять общие принципы проектирования к вашей сборке. Это даст вам идеальную структуру для сборки, так что вы сможете разработать хорошо структурированную и легко поддерживаемую, понятную сборку.

Глубокий API — Используя этот API, он позволяет вам отслеживать и настраивать его конфигурацию и поведение при выполнении.

Масштабирование Gradle — Gradle может легко увеличить свою производительность, от простых сборок одного проекта до огромных многопроектных сборок предприятия.

Многопроектные сборки — Gradle поддерживает многопроектные сборки и поддерживает частичные сборки. Если вы строите подпроект, Gradle позаботится о создании всех подпроектов, от которых он зависит.

Различные способы управления вашими сборками — Gradle поддерживает различные стратегии для управления вашими зависимостями.

Gradle — это первый инструмент интеграции сборки — Gradle полностью поддерживается для задач ANT, инфраструктура репозитория Maven и lvy для публикации и получения

зависимостей. Gradle также предоставляет конвертер для превращения Maven pom.xml в скрипт Gradle.

Легкость миграции — Gradle может легко адаптироваться к любой вашей структуре. Поэтому вы всегда можете разработать свою сборку Gradle в той же ветке, где вы можете создать живой скрипт.

Gradle Wrapper — Gradle Wrapper позволяет выполнять сборки Gradle на машинах, где Gradle не установлен. Это полезно для непрерывной интеграции серверов.

Бесплатный открытый исходный код — Gradle — это проект с открытым исходным кодом, который распространяется под лицензией Apache Software License (ASL).

Groovy — скрипт сборки Gradle написан на Groovy. Весь дизайн Gradle ориентирован на использование в качестве языка, а не жесткой структуры. А Groovy позволяет вам написать собственный скрипт с некоторыми абстракциями. Весь API Gradle полностью разработан на языке Groovy.

Почему Groovy?

Полный API Gradle разработан с использованием языка Groovy. Это преимущество внутреннего DSL над XML. Gradle — это универсальный инструмент для сборки; основное внимание уделяется Java-проектам. В таких проектах члены команды будут очень хорошо знакомы с Java, и лучше, чтобы сборка была максимально прозрачной для всех членов команды.

Такие языки, как Python, Groovy или Ruby, лучше подходят для сборки фреймворка. Почему Groovy был выбран, так это потому, что он предлагает наибольшую прозрачность для людей, использующих Java. Базовый синтаксис Groovy такой же, как Java. Groovy предлагает гораздо больше.

Gradle — Установка

Gradle — это инструмент для сборки, основанный на Java. Есть некоторые предварительные условия, которые должны быть установлены перед установкой рамы Gradle.

Предпосылки

JDK и Groovy являются необходимыми условиями для установки Gradle.

- Gradle требует JDK версии 6 или более поздней версии для установки в вашей системе. Он использует библиотеки JDK, которые установлены и установлены в переменную окружения JAVA_HOME.

- Gradle содержит собственную библиотеку Groovy, поэтому нам не нужно явно устанавливать Groovy. Если он установлен, Gradle игнорирует его.

Gradle требует JDK версии 6 или более поздней версии для установки в вашей системе. Он использует библиотеки JDK, которые установлены и установлены в переменную окружения JAVA_HOME.

Gradle содержит собственную библиотеку Groovy, поэтому нам не нужно явно устанавливать Groovy. Если он установлен, Gradle игнорирует его.

Ниже приведены инструкции по установке Gradle в вашей системе.

Шаг 1 — Проверьте установку JAVA

Прежде всего, вам необходимо установить Java Software Development Kit (SDK) в вашей системе. Чтобы убедиться в этом, выполните команду **Java –version** на любой платформе, с которой вы работаете.

В винде —

Выполните следующую команду, чтобы проверить установку Java. Я установил JDK 1.8 в моей системе.

```
C:\> java - version
```

Выход —

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)
```

В Linux —

Выполните следующую команду, чтобы проверить установку Java. Я установил JDK 1.8 в моей системе.

```
$ java - version
```

Выход —

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)
```

Мы предполагаем, что читатели этого руководства установили Java SDK версии 1.8.0_66 в своей системе.

Шаг 2 — Загрузите файл сборки Gradle

Загрузите последнюю версию Gradle по ссылке Download Gradle . На странице ссылок нажмите на ссылку **Полный дистрибутив** . Этот шаг является общим для любой платформы. Для этого вы получите полный дистрибутивный файл в папку «Загрузки».

Шаг 3 — Настройка среды для Gradle

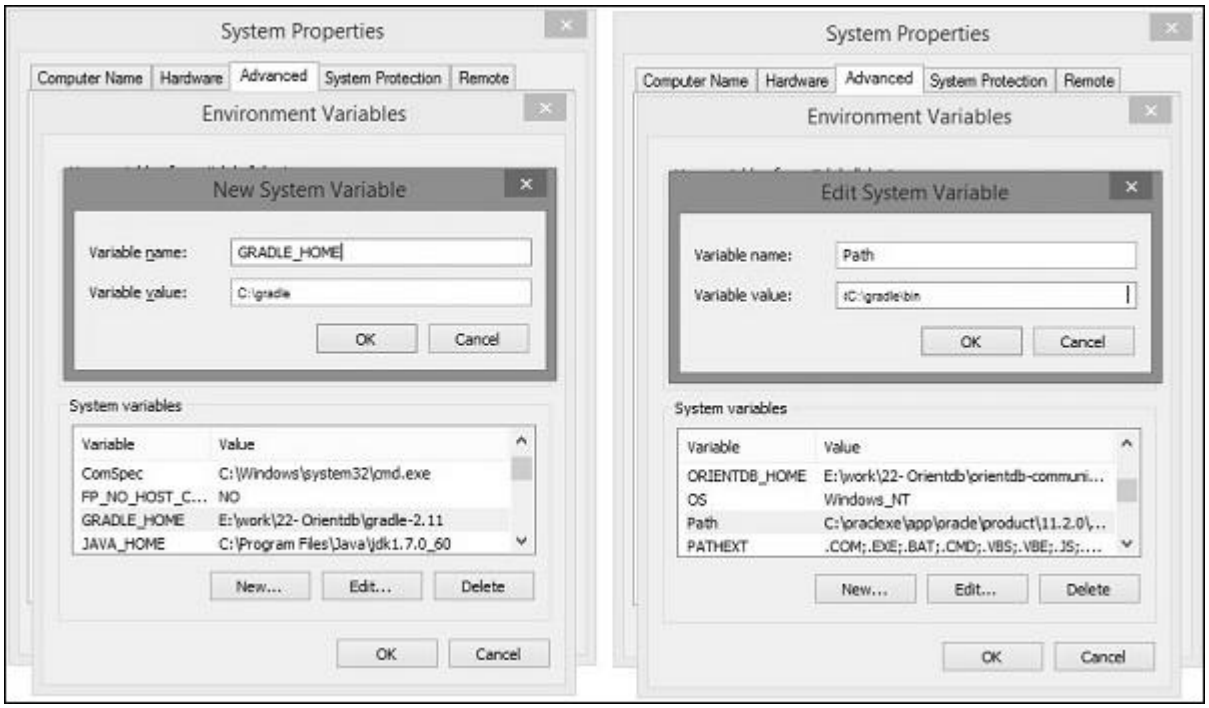
Настройка среды означает, что мы должны извлечь дистрибутивный файл, скопировать файлы библиотеки в нужное место. Настройка **переменных** среды **GRADLE_HOME** и **PATH** .

Этот шаг зависит от платформы.

В винде —

Извлеките загруженный zip-файл с именем **gradle-2.11-all.zip** и скопируйте дистрибутивные файлы из **каталога Downloads \ gradle-2.11 ** в **C: \ gradle ** location.

После этого добавьте **каталоги C: \ gradle** и **C: \ gradle \ bin** в системные переменные **GRADLE_HOME** и **PATH** . Следуйте приведенным инструкциям, **щелкнув правой кнопкой мыши на моих компьютерах -> выберите свойства -> дополнительные параметры системы -> нажмите переменные среды** . Там вы найдете диалоговое окно для создания и редактирования системных переменных. Нажмите на новую кнопку для создания переменной GRADLE_HOME (следуйте скриншоту слева). Нажмите на Edit для редактирования существующей системной переменной Path (следуйте скриншоту справа). Следуйте приведенным ниже скриншотам.



В Linux —

Распакуйте загруженный zip-файл с именем **gradle-2.11-all.zip,** после чего вы найдете извлеченный файл с именем **gradle-2.11** .

Вы можете использовать следующее, чтобы переместить дистрибутивные файлы из **Downloads / gradle-2.11 /** в **/ opt / gradle /** location. Выполните эту операцию из каталога загрузок.

```
$ sudo mv gradle-2.11 /opt/gradle
```

Отредактируйте файл ~ / .bashrc, вставьте в него следующее содержимое и сохраните его.

```
export ORIENT_HOME = /opt/gradle
export PATH = $PATH:
```

Выполните следующую команду, чтобы выполнить файл **~ / .bashrc** .

```
$ source ~/.bashrc
```

Шаг 4: Проверьте установку Gradle

В окнах:

Вы можете выполнить следующую команду в командной строке.

```
C:\> gradle –v
```

Вывод: там вы найдете версию Gradle.

```
------------------------------------------------------------
Gradle 2.11
------------------------------------------------------------

Build time: 2016-02-08 07:59:16 UTC
Build number: none
Revision: 584db1c7c90bdd1de1d1c4c51271c665bfcba978
Groovy: 2.4.4

Ant: Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM: 1.7.0_60 (Oracle Corporation 24.60-b09)
OS: Windows 8.1 6.3 amd64
```

В Linux:

Вы можете выполнить следующую команду в терминале.

```
$ gradle –v
```

Вывод: там вы найдете версию Gradle.

```
------------------------------------------------------------
Gradle 2.11
------------------------------------------------------------

Build time: 2016-02-08 07:59:16 UTC
```

```
Build number: none
Revision: 584db1c7c90bdd1de1d1c4c51271c665bfcba978


Groovy: 2.4.4
Ant: Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM: 1.7.0_60 (Oracle Corporation 24.60-b09)
OS: Linux 3.13.0-74-generic amd64
```

Gradle — Build Script

Gradle создает файл сценария для обработки двух вещей; один — это **проекты,** а другой — **задачи** . Каждая сборка Gradle представляет один или несколько проектов. Проект представляет собой библиотечный JAR или веб-приложение или может представлять собой ZIP, собранный из JAR, созданных другими проектами. Проще говоря, проект состоит из разных задач. Задача означает часть работы, которую выполняет сборка. Задачей может быть компиляция некоторых классов, создание JAR, генерация Javadoc или публикация некоторых архивов в хранилище.

Gradle использует **Groovy** для написания скриптов.

Написание сценария сборки

Gradle предоставляет предметно-ориентированный язык (DSL) для описания сборок. Это использует язык Groovy, чтобы упростить описание сборки. Каждый сценарий сборки Gradle кодируется с использованием UTF-8, сохраняется в автономном режиме и называется build.gradle.

build.gradle

Мы описываем задачи и проекты с помощью скрипта Groovy. Вы можете запустить сборку Gradle с помощью команды Gradle. Эта команда ищет файл с именем **build.gradle** . Взгляните на следующий пример, представляющий небольшой скрипт, который печатает **tutorialspoint** . Скопируйте и сохраните следующий скрипт в файл с именем **build.gradle** . Этот скрипт сборки определяет имя задачи hello, которое используется для вывода строки tutorialspoint.

```
task hello {
   doLast {
      println 'tutorialspoint'
   }
}
```

Выполните следующую команду в командной строке. Он выполняет вышеуказанный скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle -q hello
```

Выход:

```
tutorialspoint
```

Если вы думаете, что задача работает аналогично цели ANT, то это правильно — задача Gradle эквивалентна цели ANT.

Вы можете упростить эту задачу приветствия, указав ярлык (представляет символ <<) для оператора **doLast** . Если вы добавите этот ярлык к вышеупомянутой задаче, **привет,** он будет выглядеть как следующий скрипт.

```
task hello << {
   println 'tutorialspoint'
}
```

Как и выше, вы можете выполнить приведенный выше скрипт с помощью команды **gradle –q hello** .

Сценарий Grade в основном использовал два реальных объекта, один из них — объект проекта, а другой — объект сценария.

Объект проекта — каждый сценарий описывает один или несколько проектов. Во время выполнения этот сценарий настраивает объект проекта. Вы можете вызывать некоторые методы и использовать свойства в вашем скрипте сборки, которые делегированы объекту проекта.

Сценарий Объект — Gradle взят код сценария в классы, который реализует интерфейс сценариев, а затем выполняется. Это означает, что все свойства и методы, объявленные интерфейсом скрипта, доступны в вашем скрипте.

В следующей таблице приведен список **стандартных свойств проекта** . Все эти свойства доступны в вашем скрипте сборки.

Старший	название	Тип	Значение по умолчанию
1	проект	проект	Экземпляр проекта
2	название	строка	Название каталога проекта.
3	дорожка	строка	Абсолютный путь проекта.
4	описание	строка	Описание для проекта.

5	ProjectDir	файл	Каталог, содержащий скрипт сборки.
6	buildDir	файл	ProjectDir / сборки
7	группа	объект	Неопределенные
8	версия	объект	Неопределенные
9	муравей	AntBuilder	Экземпляр AntBuilder

Groovy Основы

Скрипты сборки Gradle используют полнофункциональный Groovy API. В качестве стартапа взгляните на следующие примеры.

В следующем примере объясняется, как преобразовать строку в верхний регистр.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** .

```
task upper << {
   String expString = 'TUTORIALS point'
   println "Original: " + expString
   println "Upper case: " + expString.toUpperCase()
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle -q upper
```

Выход:

```
Original: TUTORIALS point
Upper case: TUTORIALS POINT
```

В следующем примере объясняется, как печатать значение неявного параметра ($ it) 4 раза.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task count << {
   4.times {
      print "$it "
```

```
   }
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
$ gradle –q count
```

Выход:

```
0 1 2 3
```

Groovy язык предоставляет множество функций, некоторые из которых обсуждаются ниже.

Groovy JDK Методы

Groovy добавляет множество полезных методов в стандартные классы Java. Например, Iterable API из JDK реализует метод **each (),** который выполняет итерацию по элементам Iterable Interface.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task groovyJDK << {
   String myName = "Marc";
   myName.each() {
      println "${it}"
   };
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle –q groovyJDK
```

Выход:

```
M
a
r
c
```

Средства доступа к недвижимости

Вы можете автоматически получить доступ к соответствующим методам получения и установки определенного свойства, указав его ссылку.

Следующий фрагмент определяет синтаксис методов getter и setter свойства **buildDir** .

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

Необязательные скобки при вызове метода

Groovy содержит специальную функцию в вызове методов, которая является необязательной скобкой для вызова метода. Эта функция также применима к сценариям Gradle.

Посмотрите на следующий синтаксис. Это определяет метод,
вызывающий **systemProperty тестового** объекта.

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

Закрытие как последний параметр метода

Gradle DSL использует замыкания во многих местах. Если последний параметр метода является закрытием, вы можете поместить закрытие после вызова метода.

Следующий фрагмент кода определяет синтаксис, используемый Closures в качестве параметров метода repositories ().

```
repositories {
   println "in a closure"
}
repositories() {
   println "in a closure"
}
repositories({ println "in a closure" })
```

Импорт по умолчанию

Gradle автоматически добавляет набор операторов импорта в сценарии Gradle. В следующем списке показаны пакеты импорта по умолчанию для скрипта Gradle.

Ниже приведены стандартные пакеты импорта в скрипт Gradle.

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.cache.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.component.*
import org.gradle.api.credentials.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.dsl.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.logging.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.jetty.*
import org.gradle.api.plugins.osgi.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.plugins.sonar.*
import org.gradle.api.plugins.sonar.model.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
```

```
import org.gradle.api.publish.plugins.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.model.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.authentication.*
import org.gradle.authentication.http.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.external.javadoc.*
import org.gradle.ide.cdt.*
import org.gradle.ide.cdt.tasks.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.application.scripts.*
import org.gradle.jvm.application.tasks.*
import org.gradle.jvm.platform.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.tasks.api.*
```

```
import org.gradle.jvm.test.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.plugins.*
import org.gradle.language.base.sources.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.coffeescript.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.java.tasks.*
import org.gradle.language.javascript.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.routes.*
import org.gradle.language.scala.*
import org.gradle.language.scala.plugins.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.scala.toolchain.*
```

```
import org.gradle.language.twirl.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.googletest.*
import org.gradle.nativeplatform.test.googletest.plugins.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.platform.base.test.*
import org.gradle.play.*
import org.gradle.play.distribution.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
import org.gradle.plugins.javascript.envjs.http.simple.*
```

```
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
import org.gradle.plugins.javascript.rhino.worker.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.sonar.runner.*
import org.gradle.sonar.runner.plugins.*
import org.gradle.sonar.runner.tasks.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.testkit.runner.*
import org.gradle.util.*
```

Gradle — Задачи

Скрипт сборки Gradle описывает один или несколько проектов. Каждый проект состоит из разных задач. Задача — это часть работы, которую выполняет сборка. Задачей может быть компиляция некоторых классов, хранение файлов классов в отдельной целевой папке, создание JAR, генерация Javadoc или публикация некоторых достижений в репозитории.

В этой главе объясняется, что такое задача, а также как ее создать и выполнить.

Определение задач

Задача — это ключевое слово, которое используется для определения задачи в сценарии сборки. Посмотрите на следующий пример, который представляет задачу с именем **hello,** которая печатает **tutorialspoint** . Скопируйте и сохраните следующий скрипт в файл с именем **build.gradle** . Этот сценарий сборки определяет имя задачи hello, которое используется для печати строки tutorialspoint.

```
task hello {
   doLast {
      println 'tutorialspoint'
   }
}
```

Выполните следующую команду в командной строке. Он выполняет вышеуказанный скрипт. Вы должны выполнить это там, где хранится файл build.gradle.

```
C:\> gradle –q hello
```

Выход:

```
tutorialspoint
```

Вы можете упростить эту задачу приветствия, указав ярлык (представляет символ <<) для оператора **doLast** . Если вы добавите этот ярлык к вышеупомянутой задаче, **привет,** он будет выглядеть как следующий скрипт.

```
task hello << {
   println 'tutorialspoint'
}
```

Вы можете выполнить приведенный выше скрипт с помощью команды **gradle –q hello** .

Вот несколько вариантов определения задачи, посмотрите на нее. В следующем примере определяется задача **hello** .

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task (hello) << {
   println "tutorialspoint"
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle –q hello
```

Выход:

```
tutorialspoint
```

Вы также можете использовать строки для имен задач. Взгляните на тот же самый привет пример. Здесь мы будем использовать String как задачу.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task('hello') << {
   println "tutorialspoint"
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle -q hello
```

Выход:

```
tutorialspoint
```

Вы также можете использовать альтернативный синтаксис для определения задачи. Это использует метод create () для определения задачи. Взгляните на тот же самый привет пример, приведенный ниже.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** .

```
tasks.create(name: 'hello') << {
   println "tutorialspoint"
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle -q hello
```

Выход:

```
tutorialspoint
```

Поиск задач

Если вы хотите найти задачи, которые вы определили в файле сборки, вам нужно использовать соответствующие стандартные свойства проекта. Это означает, что каждая задача доступна как свойство проекта, используя имя задачи в качестве имени свойства.

Взгляните на следующий код, который обращается к задачам как к свойствам.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** .

```
task hello


println hello.name
println project.hello.name
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle -q hello
```

Выход:

```
hello
hello
```

Вы также можете использовать все свойства через коллекцию задач.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task hello


println tasks.hello.name
println tasks['hello'].name
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle –q hello
```

Выход:

```
hello
hello
```

Вы также можете получить доступ к пути задачи, используя задачи. Для этого вы можете вызвать метод getByPath () с именем задачи, либо относительным путем, либо абсолютным путем.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** .

```
project(':projectA') {
   task hello
}
task hello


println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где хранится файл build.gradle.

```
C:\> gradle –q hello
```

Выход:

```
:hello
```

```
:hello
:projectA:hello
:projectA:hello
```

Добавление зависимостей к задачам

Вы можете сделать задачу зависимой от другой задачи, что означает, что когда одна задача выполнена, тогда запускается только другая задача. Каждое задание отличается от имени задания. Коллекция имен задач называется ее коллекцией задач. Чтобы сослаться на задачу в другом проекте, вы должны использовать путь к проекту в качестве префикса к соответствующему имени задачи.

Следующий пример, который добавляет зависимость из taskX в taskY.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** . Посмотрите на следующий код.

```
task taskX << {
   println 'taskX'
}
task taskY(dependsOn: 'taskX') << {
   println "taskY"
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где **хранится** файл **build.gradle** .

```
C:\> gradle –q taskY
```

Выход:

```
taskX
taskY
```

Приведенный выше пример добавляет зависимость от задачи, используя ее имена. Существует еще один способ достижения зависимости задачи, который заключается в определении зависимости с помощью объекта Task.

Давайте возьмем тот же пример зависимости задачиY от задачи X, но мы используем объекты задачи вместо имен ссылок на задачи.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task taskY << {
   println 'taskY'
}
```

```
task taskX << {
   println 'taskX'
}
taskY.dependsOn taskX
```

Выполните следующую команду в командной строке. Вы должны выполнить это там, где хранится файл build.gradle.

```
C:\> gradle –q taskY
```

Выход:

```
taskX
taskY
```

Приведенный выше пример добавляет зависимость от задачи, используя ее имена. Есть еще один способ достижения зависимости задачи — определить зависимость с помощью объекта Task.

Здесь мы берем тот же пример, что taskY зависит от taskX, но мы используем объекты задачи вместо имен ссылок на задачи. Взгляните на это.

Скопируйте и сохраните приведенный ниже код в файл **build.gradle** . Посмотрите на следующий код.

```
task taskX << {
   println 'taskX'
}
taskX.dependsOn {
   tasks.findAll {
      task → task.name.startsWith('lib')
   }
}
task lib1 << {
   println 'lib1'
}
task lib2 << {
   println 'lib2'
}
task notALib << {
   println 'notALib'
}
```

Выполните следующую команду в командной строке. Он выполняет приведенный выше скрипт. Вы должны выполнить это, где **хранится** файл **build.gradle** .

```
C:\> gradle –q taskX
```

Выход:

```
lib1
lib2
taskX
```

Добавление описания к задаче

Вы можете добавить описание к вашей задаче. Это описание отображается при выполнении **задач Gradle** . Это возможно с помощью ключевого слова description.

Скопируйте и сохраните следующий код в файл **build.gradle** . Посмотрите на следующий код.

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
   println("description applied")
}
```

Выполните следующую команду в командной строке. Вы должны выполнить это там, где хранится файл build.gradle.

```
C:\> gradle –q copy
```

Если команда выполнена успешно, вы получите следующий вывод.

```
description applied
```

Пропуск задач

Пропуск задач может быть выполнен путем передачи предиката замыкания. Это возможно только в том случае, если метод задачи или замыкание **генерируют исключение StopExecutionException** до выполнения фактической работы задачи.

Скопируйте и сохраните следующий код в файл **build.gradle** .

```
task eclipse << {
   println 'Hello Eclipse'
}
```

```
// #1st approach - closure returning true, if the task should be executed, false if not.
eclipse.onlyIf {
   project.hasProperty('usingEclipse')
}

// #2nd approach - alternatively throw an StopExecutionException() like this
eclipse.doFirst {
   if(!usingEclipse) {
      throw new StopExecutionException()
   }
}
```

Выполните следующую команду в командной строке. Вы должны выполнить это там, где хранится файл build.gradle.

```
C:\> gradle –q eclipse
```

Структура задачи

Gradle имеет разные фазы при работе с заданиями. Прежде всего, это фаза конфигурации, на которой выполняется код, который указывается непосредственно при закрытии задачи. Блок конфигурации выполняется для каждой доступной задачи, а не только для тех задач, которые впоследствии фактически выполняются.

После фазы конфигурирования фаза выполнения запускает код внутри замыканий **doFirst** или **doLast** тех задач, которые фактически выполняются.

Gradle — управление зависимостями

Скрипт сборки Gradle определяет процесс сборки проектов; каждый проект содержит некоторые зависимости и некоторые публикации. Зависимости означают вещи, которые поддерживают создание вашего проекта, такие как требуемый файл JAR из других проектов и внешние JAR, такие как JDBC JAR или Eh-cache JAR в пути к классам. Публикации означают результаты проекта, такие как файлы тестовых классов и файлы сборки, такие как файлы war.

Все большинство всех проектов не являются самостоятельными. Им нужны файлы, созданные другими проектами для компиляции и тестирования исходных файлов. Например, чтобы использовать Hibernate в проекте, вам нужно включить некоторые JAR-файлы Hibernate в classpath. Gradle использует специальный скрипт для определения зависимостей, которые необходимо загрузить.

Gradle позаботится о создании и публикации результатов где-нибудь. Публикация основана на задаче, которую вы определяете. Возможно, вы захотите скопировать файлы в локальный каталог или загрузить их в удаленный репозиторий Maven или Ivy, или вы можете использовать файлы из другого проекта в той же многопроектной сборке. Мы можем назвать процесс публикации задачи публикацией.

Объявление ваших зависимостей

Конфигурация зависимостей — это не что иное, как набор множеств зависимостей. Вы можете использовать эту функцию для объявления внешних зависимостей средствами, которые вы хотите загрузить из Интернета. Это определяет различные стандарты, такие как следующие.

```
apply plugin: 'java'


repositories {
   mavenCentral()
}


dependencies {
   compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
   testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

Конфигурации зависимостей

Конфигурация зависимостей — это не что иное, как набор зависимостей. Вы можете использовать эту функцию для объявления внешних зависимостей, которые вы хотите загрузить из Интернета. Это определяет следующие различные стандартные конфигурации.

- **Компилировать** — зависимости, необходимые для компиляции производственного источника проекта.
- **Runtime** — зависимости, необходимые производственным классам во время выполнения. По умолчанию также включает зависимости времени компиляции.
- **Тестовая компиляция** — зависимости, необходимые для компиляции исходного кода проекта. По умолчанию он включает скомпилированные производственные классы и зависимости времени компиляции.
- **Test Runtime** — зависимости, необходимые для запуска тестов. По умолчанию он включает зависимости времени выполнения и тестовой компиляции.

Компилировать — зависимости, необходимые для компиляции производственного источника проекта.

Runtime — зависимости, необходимые производственным классам во время выполнения. По умолчанию также включает зависимости времени компиляции.

Тестовая компиляция — зависимости, необходимые для компиляции исходного кода проекта. По умолчанию он включает скомпилированные производственные классы и зависимости времени компиляции.

Test Runtime — зависимости, необходимые для запуска тестов. По умолчанию он включает зависимости времени выполнения и тестовой компиляции.

Внешние зависимости

Внешние зависимости относятся к типу зависимостей. Это зависимость от некоторых файлов, созданных вне текущей сборки и хранящихся в каком-либо репозитории, таком как Maven central, или в корпоративном репозитории Maven или lvy, или в каталоге в локальной файловой системе.

Следующий фрагмент кода предназначен для определения внешней зависимости. Используйте этот код в файле **build.gradle** .

```
dependencies {
   compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
}
```

Внешняя зависимость объявляет внешние зависимости, а форма ярлыка выглядит как «группа: имя: версия».

Хранилища

При добавлении внешних зависимостей. Gradle ищет их в хранилище. Репозиторий — это просто набор файлов, упорядоченный по группам, именам и версиям. По умолчанию Gradle не определяет никаких репозиториев. Мы должны определить хотя бы одно хранилище явно. Следующий фрагмент кода определяет, как определить хранилище maven. Используйте этот код в файле **build.gradle** .

```
repositories {
   mavenCentral()
}
```

Следующий код предназначен для определения удаленного Maven. Используйте этот код в файле **build.gradle** .

```
repositories {
   maven {
      url "http://repo.mycompany.com/maven2"
   }
}
```

Издательские артефакты

Конфигурации зависимостей также используются для публикации файлов. Эти опубликованные файлы называются артефактами. Обычно мы используем плагины для определения артефактов. Тем не менее, вам нужно указать Gradle, где публиковать артефакты. Вы можете достичь этого, прикрепив репозитории к задаче загрузки архивов. Взгляните на следующий

синтаксис публикации репозитория Maven. При выполнении Gradle создаст и загрузит Pom.xml в соответствии с требованиями проекта. Используйте этот код в файле **build.gradle** .

```
apply plugin: 'maven'


uploadArchives {
   repositories {
      mavenDeployer {
         repository(url: "file://localhost/tmp/myRepo/")
      }
   }
}
```

Gradle — Плагины

Плагин — это не что иное, как набор задач, почти все полезные задачи, такие как компиляция задач, настройка объектов домена, настройка исходных файлов и т. Д., Выполняются плагинами. Применение плагина к проекту означает, что он позволяет расширять возможности проекта. Плагины могут делать такие вещи, как —

- Расширьте базовую модель Gradle (например, добавьте новые элементы DSL, которые можно настроить).
- Настройте проект в соответствии с преобразованиями (например, добавьте новые задачи или настройте разумные значения по умолчанию).
- Применить конкретную конфигурацию (например, добавить организационные репозитории или обеспечить соблюдение стандартов).

Типы плагинов

В Gradle есть два типа плагинов: плагины для скриптов и бинарные плагины. Плагины скриптов — это дополнительный скрипт сборки, который дает декларативный подход к манипулированию сборкой. Обычно это используется в сборке. Бинарные плагины — это классы, которые реализуют интерфейс плагина и применяют программный подход к управлению сборкой. Бинарные плагины могут находиться со скриптом сборки, с иерархией проекта или внешне в JAR плагина.

Применение плагинов

Метод API **Project.apply ()** используется для применения определенного плагина. Вы можете использовать один и тот же плагин несколько раз. Существует два типа плагинов: один — плагин скрипта, а второй — бинарный плагин.

Скриптовые плагины

Плагины сценариев могут быть применены из сценария в локальной файловой системе или в удаленном месте. Расположение файловой системы относится к каталогу проекта, в то время как расположение удаленного сценария указывает URL-адрес HTTP. Взгляните на следующий

фрагмент кода. Он используется для применения плагина **other.gradle** к сценарию сборки. Используйте этот код в файле **build.gradle** .

```
apply from: 'other.gradle'
```

Бинарные плагины

Каждый плагин идентифицируется по идентификатору плагина, так как некоторые основные плагины используют короткие имена для его применения, а некоторые плагины сообщества используют полное имя для идентификатора плагина. Некоторое время это позволяет указать класс плагина.

Взгляните на следующий фрагмент кода. Он показывает, как применить плагин Java, используя его тип. Используйте этот код в файле **build.gradle** .

```
apply plugin: JavaPlugin
```

Посмотрите на следующий код для применения основного плагина, используя короткое имя. Используйте этот код в файле **build.gradle** .

```
plugins {
   id 'java'
}
```

Взгляните на следующий код для применения плагина сообщества с использованием короткого имени. Используйте этот код в файле **build.gradle** .

```
plugins {
   id "com.jfrog.bintray" version "0.4.1"
}
```

Написание пользовательских плагинов

При создании пользовательского плагина вам необходимо написать реализацию плагина. Gradle создает плагин и вызывает экземпляр плагина с помощью метода Plugin.apply (). Следующий пример содержит плагин приветствия, который добавляет задачу приветствия в проект. Посмотрите на следующий код. Используйте этот код в файле **build.gradle** .

```
apply plugin: GreetingPlugin


class GreetingPlugin implements Plugin<Project> {
   void apply(Project project) {
      project.task('hello') << {
         println "Hello from the GreetingPlugin"
      }
```

```
   }
}
```

Используйте следующий код для выполнения вышеуказанного сценария.

```
C:\> gradle -q hello
```

Выход:

```
Hello from the GreetingPlugin
```

Получение информации от сборки

Большинству плагинов требуется поддержка конфигурации из скрипта сборки. У проекта Gradle есть связанный объект ExtensionContainer, который помогает отслеживать все настройки и свойства, передаваемые плагинам.

Давайте добавим простой объект расширения в проект. Здесь мы добавляем объект расширения приветствия в проект, который позволяет настроить приветствие. Используйте этот код в файле **build.gradle** .

```
apply plugin: GreetingPlugin

greeting.message = 'Hi from Gradle'

class GreetingPlugin implements Plugin<Project> {
   void apply(Project project) {
      // Add the 'greeting' extension object
      project.extensions.create("greeting", GreetingPluginExtension)

      // Add a task that uses the configuration
      project.task('hello') << {
         println project.greeting.message
      }
   }
}

class GreetingPluginExtension {
   def String message = 'Hello from GreetingPlugin'
}
```

Используйте следующий код для выполнения вышеуказанного сценария.

```
C:\> gradle -q hello
```

Выход:

```
Hi from Gradle
```

В этом примере GreetingPlugin — это простой старый объект Groovy с полем с именем message. Объект расширения добавляется в список плагинов с именем приветствия. Затем этот объект становится доступным как свойство проекта с тем же именем, что и у объекта расширения.

Gradle добавляет закрытие конфигурации для каждого объекта расширения, поэтому вы можете сгруппировать настройки вместе. Посмотрите на следующий код. Используйте этот код в файле **build.gradle** .

```
apply plugin: GreetingPlugin

greeting {
   message = 'Hi'
   greeter = 'Gradle'
}

class GreetingPlugin implements Plugin<Project> {
   void apply(Project project) {
      project.extensions.create("greeting", GreetingPluginExtension)

      project.task('hello') << {
         println "${project.greeting.message} from ${project.greeting.greeter}"
      }
   }
}

class GreetingPluginExtension {
   String message
   String greeter
}
```

Используйте следующий код для выполнения вышеуказанного сценария.

```
C:\> gradle -q hello
```

Выход:

```
Hello from Gradle
```

Стандартные плагины Gradle

Существуют различные плагины, которые включены в дистрибутив Gradle.

Языковые плагины

Эти плагины добавляют поддержку различных языков, которые могут быть скомпилированы и выполнены в JVM.

Идентификатор плагина	Автоматически применяется	Описание
Джава	Java-база	Добавляет в проект возможности компиляции, тестирования и связывания Java. Он служит основой для многих других плагинов Gradle.
заводной	Java, заводная база	Добавлена поддержка построения Groovy проектов.
Скала	Java, Scala-база	Добавлена поддержка для построения проектов Scala.
ANTLR	Джава	Добавлена поддержка генерации парсеров с использованием Antlr.

Инкубация языковых плагинов

Эти плагины добавляют поддержку различных языков.

Идентификатор плагина	Автоматически применяется	Описание
ассемблер	—	Добавляет возможности родного языка ассемблера в проект.
с	—	Добавляет возможности компиляции исходного кода C в проект.
CPP	—	Добавляет возможности компиляции исходного кода C ++ в проект.
Objective-C	—	Добавляет возможности компиляции исходного кода Objective-C в проект.

Цель-каст	—	Добавляет в проект возможности компиляции исходного кода Objective-C ++.
окна-ресурсы	—	Добавлена поддержка включения ресурсов Windows в собственные двоичные файлы.

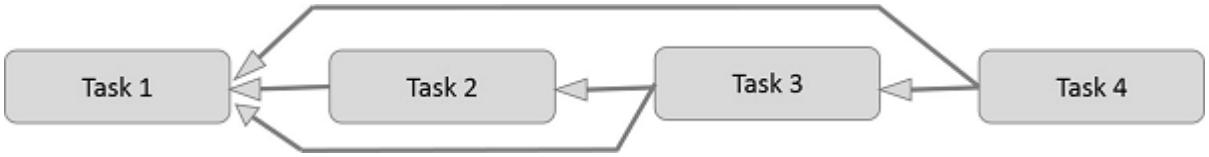
Gradle — Запуск сборки

Gradle предоставляет командную строку для выполнения сценария сборки. Он может выполнять более одной задачи одновременно. В этой главе объясняется, как выполнять несколько задач с использованием разных параметров.

Выполнение нескольких задач

Вы можете выполнить несколько задач из одного файла сборки. Gradle может обработать этот файл сборки с **помощью команды gradle** . Эта команда скомпилирует каждую задачу в порядке их перечисления и выполнит каждую задачу вместе с зависимостями, используя различные параметры.

Пример — есть четыре задачи — задача1, задача2, задача3 и задача4. Задача 3 и задача 4 зависят от задачи 1 и задачи 2. Посмотрите на следующую диаграмму.



В приведенном выше 4 задачи зависят друг от друга и представлены символом стрелки. Посмотрите на следующий код. Копировать можно вставить в файл **build.gradle** .

```
task task1 << {
   println 'compiling source'
}

task task2(dependsOn: task1) << {
   println 'compiling unit tests'
}

task task3(dependsOn: [task1, task2]) << {
   println 'running unit tests'
}

task task4(dependsOn: [task1, task3]) << {
   println 'building the distribution'
```

```
}
```

Вы можете использовать следующий код для компиляции и выполнения вышеуказанной задачи.

```
C:\> gradle task4 test
```

Выход:

```
:task1
compiling source
:task2
compiling unit tests
:task3
running unit tests
:task4
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

Исключая задачи

При исключении задачи из выполнения вы можете использовать опцию –x вместе с командой gradle и указать имя задачи, которое вы хотите исключить.

Используйте следующую команду, чтобы исключить task4 из приведенного выше сценария.

```
C:\> gradle task4 -x test
```

Выход:

```
:task1
compiling source
:task4
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

Продолжение сборки, когда происходит сбой

Gradle прервет выполнение и завершит сборку сразу после сбоя любой задачи. Вы можете продолжить выполнение даже в случае сбоя. Для этого вы должны использовать опцию –continue с командой gradle. Он обрабатывает каждую задачу отдельно вместе с их зависимостями. И главное — он будет отлавливать каждую обнаруженную ошибку и сообщать об окончании сборки. Предположим, что если задача не выполнена, то последующие зависимые задачи также не будут выполнены.

Выбор сборки для выполнения

Когда вы запускаете команду gradle, она ищет файл сборки в текущем каталоге. Вы можете использовать опцию –b, чтобы выбрать конкретный файл сборки вместе с абсолютным путем. В следующем примере выбирается проект hello из файла **myproject.gradle,** который находится в **подкаталоге /** посмотрите на него.

```
task hello << {
   println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

Вы можете использовать следующую команду для выполнения вышеуказанного скрипта.

```
C:\> gradle -q -b subdir/myproject.gradle hello
```

Выход:

```
using build file 'myproject.gradle' in 'subdir'.
```

Получение информации о сборке

Gradle предоставляет несколько встроенных задач для получения подробной информации о задаче и проекте. Это может быть полезно для понимания структуры и зависимостей вашей сборки и для устранения проблем. Вы можете использовать плагин отчетов проекта, чтобы добавить задачи в ваш проект, который будет генерировать эти отчеты.

Листинг проектов

Вы можете перечислить иерархию проектов выбранного проекта и его подпроектов с помощью команды **gradle –q projects** . Вот пример, используйте следующую команду, чтобы получить список всех проектов в файле сборки.

```
C:\> gradle -q projects
```

Выход:

```
------------------------------------------------------------
Root project
------------------------------------------------------------
```

```
Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

Отчет показывает описание каждого проекта, если он указан. Вы можете использовать следующую команду, чтобы указать описание. Вставьте его в файл **build.gradle** .

```
description = 'The shared API for the application'
```

Листинг Задачи

Вы можете перечислить все задачи, которые принадлежат нескольким проектам, используя следующую команду.

```
C:\> gradle -q tasks
```

Выход:

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project 'projectR
eports'.
```

```
components - Displays the components produced by root project 'projectReports'. [incubating
]

dependencies - Displays all dependencies declared in root project 'projectReports'.

dependencyInsight - Displays the insight into a specific dependency in root project 'projec
tReports'.

help - Displays a help message.

model - Displays the configuration model of root project 'projectReports'. [incubating]

projects - Displays the sub-projects of root project 'projectReports'.

properties - Displays the properties of root project 'projectReports'.

tasks - Displays the tasks runnable from root project 'projectReports'

   (some of the displayed tasks may belong to subprojects).


To see all tasks and more detail, run gradle tasks --all


To see more detail about a task, run gradle help --task <task>
```

Вы можете использовать следующую команду для отображения информации обо всех задачах.

```
C:\> gradle -q tasks --all
```

Выход:

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------


Default tasks: dists


Build tasks
-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
   docs - Builds the documentation
api:libs - Builds the JAR
   api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
   webapp:compile - Compiles the source files


Build Setup tasks
-----------------
```

```
init - Initializes a new Gradle build. [incubating]

wrapper - Generates Gradle wrapper files. [incubating]


Help tasks

----------

buildEnvironment - Displays all buildscript dependencies declared in root project 'projectR
eports'.

api:buildEnvironment - Displays all buildscript dependencies declared in project ':api'.

webapp:buildEnvironment - Displays all buildscript dependencies declared in project ':webap
p'.

components - Displays the components produced by root project 'projectReports'. [incubating
]

api:components - Displays the components produced by project ':api'. [incubating]

webapp:components - Displays the components produced by project ':webapp'. [incubating]

dependencies - Displays all dependencies declared in root project 'projectReports'.

api:dependencies - Displays all dependencies declared in project ':api'.

webapp:dependencies - Displays all dependencies declared in project ':webapp'.

dependencyInsight - Displays the insight into a specific dependency in root project 'projec
tReports'.

api:dependencyInsight - Displays the insight into a specific dependency in project ':api'.

webapp:dependencyInsight - Displays the insight into a specific dependency in project ':web
app'.

help - Displays a help message.

api:help - Displays a help message.

webapp:help - Displays a help message.

model - Displays the configuration model of root project 'projectReports'. [incubating]

api:model - Displays the configuration model of project ':api'. [incubating]

webapp:model - Displays the configuration model of project ':webapp'. [incubating]

projects - Displays the sub-projects of root project 'projectReports'.

api:projects - Displays the sub-projects of project ':api'.

webapp:projects - Displays the sub-projects of project ':webapp'.

properties - Displays the properties of root project 'projectReports'.

api:properties - Displays the properties of project ':api'.

webapp:properties - Displays the properties of project ':webapp'.

tasks - Displays the tasks runnable from root project 'projectReports'

   (some of the displayed tasks may belong to subprojects).

api:tasks - Displays the tasks runnable from project ':api'.

webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

Вот некоторые списки команд в таблице описания различных опций.

Старший	**команда**	**Описание**

1	gradle –q help –task <имя задачи>	Предоставляет информацию об использовании (например, путь, тип, описание, группа) о конкретной задаче или нескольких задачах.
2	зависимости gradle –q	Предоставляет список зависимостей выбранного проекта.
3	gradle -q api: зависимости — configuration <имя задачи>	Предоставляет список ограниченных зависимостей, соответствующих конфигурации.
4	gradle –q buildEnvironment	Предоставляет список зависимостей сценария сборки.
5	gradle –q dependencyInsight	Предоставляет представление о конкретной зависимости.
6	Gradle –q свойства	Предоставляет список свойств выбранного проекта.

Gradle — Создайте проект JAVA

В этой главе рассказывается о том, как собрать проект Java с помощью файла сборки Gradle.

Прежде всего, мы должны добавить плагин Java в скрипт сборки, потому что он предоставляет задачи для компиляции исходного кода Java, запуска модульных тестов, создания Javadoc и создания файла JAR. Используйте следующую строку в файле **build.gradle** .

```
apply plugin: 'java'
```

Макет проекта Java по умолчанию

Когда вы добавляете плагин в свою сборку, он предполагает определенную настройку вашего Java-проекта (аналогично Maven). взгляните на следующую структуру каталогов.

- src / main / java содержит исходный код Java
- src / test / java содержит тесты Java

Если вы выполните эту настройку, следующий файл сборки будет достаточен для компиляции, тестирования и компоновки проекта Java.

Чтобы начать сборку, введите следующую команду в командной строке.

```
C:\> gradle build
```

Исходные наборы могут использоваться для указания другой структуры проекта. Например, источники хранятся в папке **src,** а не в **src / main / java** . Взгляните на следующую структуру каталогов.

```
apply plugin: 'java'
sourceSets {
   main {
      java {
         srcDir 'src'
      }
   }

   test {
      java {
         srcDir 'test'
      }
   }
}
```

Выполнение задачи init

Gradle еще не поддерживает несколько шаблонов проектов. Но он предлагает задачу **инициализации** для создания структуры нового проекта Gradle. Без дополнительных параметров эта задача создает проект Gradle, который содержит файлы-оболочки Gradle, файлы **build.gradle** и **settings.gradle** .

При добавлении параметра **—type** с **java-библиотекой в** качестве значения создается структура проекта java, и файл **build.gradle** содержит определенный шаблон Java с Junit. Посмотрите на следующий код для файла **build.gradle** .

```
apply plugin: 'java'

repositories {
   jcenter()
}

dependencies {
   compile 'org.slf4j:slf4j-api:1.7.12'
   testCompile 'junit:junit:4.12'
}
```

В разделе репозитории он определяет, где найти зависимости. **Jcenter** для разрешения ваших зависимостей. Раздел «Зависимости» предназначен для предоставления информации о внешних зависимостях.

Указание версии Java

Обычно проект Java имеет версию и целевую JRE, на которой он компилируется. Свойство **version** и **sourceCompatibility** можно установить в файле **build.gradle** .

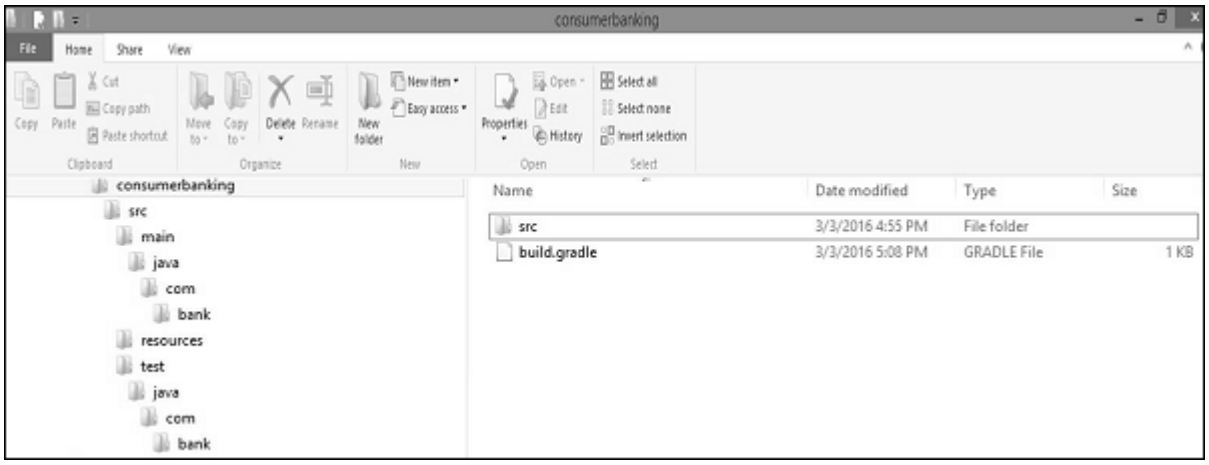
```
version = 0.1.0

sourceCompatibility = 1.8
```

Если артефакт является исполняемым Java-приложением, файл **MANIFEST.MF** должен знать класс с методом main.

```
apply plugin: 'java'


jar {
   manifest {
      attributes 'Main-Class': 'com.example.main.Application'
   }
}
```

Пример:

Создайте структуру каталогов, как показано на скриншоте ниже.



Скопируйте приведенный ниже Java-код в файл App.java и сохраните его в каталоге **consumerbanking \ src \ main \ java \ com \ bank** .

```
package com.bank;


/**
* Hello world!
*
*/


public class App {
   public static void main( String[] args ){
```

```
      System.out.println( "Hello World!" );
   }
}
```

Скопируйте приведенный ниже Java-код в файл AppTset.java и сохраните его в каталоге customerbanking **\ src \ test \ java \ com \ bank** .

```
package com.bank;

/**
* Hello world!
*
*/

public class App{
   public static void main( String[] args ){
      System.out.println( "Hello World!" );
   }
}
```

Скопируйте приведенный ниже код в файл build.gradle и поместите его в каталог **consumerbanking ** .

```
apply plugin: 'java'

repositories {
   jcenter()
}

dependencies {
   compile 'org.slf4j:slf4j-api:1.7.12'
   testCompile 'junit:junit:4.12'
}

jar {
   manifest {
      attributes 'Main-Class': 'com.example.main.Application'
   }
}
```

Для компиляции и выполнения вышеуказанного скрипта используйте приведенные ниже команды.

```
consumerbanking\> gradle tasks

consumerbanking\> gradle assemble

consumerbanking\> gradle build
```

Проверьте все файлы классов в соответствующих каталогах и проверьте папку **consumerbanking \ build \ lib на** наличие файла **consumerbanking.jar** .

Gradle — построить Groovy проект

В этой главе объясняется, как скомпилировать и выполнить проект Groovy с использованием файла **build.gradle** .

Groovy Плагин

Подключаемый модуль Groovy для Gradle расширяет подключаемый модуль Java и предоставляет задачи для программ Groovy. Вы можете использовать следующую строку для применения Groovy плагин.

```
apply plugin: 'groovy'
```

Полный файл сценария сборки выглядит следующим образом. Скопируйте следующий код в файл **build.gradle** .

```
apply plugin: 'groovy'


repositories {
   mavenCentral()
}


dependencies {
   compile 'org.codehaus.groovy:groovy-all:2.4.5'
   testCompile 'junit:junit:4.12'
}
```

Вы можете использовать следующую команду для выполнения сценария сборки.

```
gradle build
```

Макет проекта Groovy по умолчанию

Плагин Groovy предполагает определенную настройку вашего проекта Groovy.

- src / main / groovy содержит исходный код Groovy
- src / test / groovy содержит Groovy тесты
- src / main / java содержит исходный код Java

- src / test / java содержит тесты Java

Проверьте соответствующий каталог, где **находится** файл **build.gradle** для папки сборки.

Gradle — Тестирование

Тестовое задание автоматически обнаруживает и выполняет все модульные тесты в наборе исходных текстов теста. Он также генерирует отчет после завершения теста. JUnit и TestNG являются поддерживаемыми API.

Тестовое задание предоставляет метод **Test.getDebug (),** который можно настроить на запуск, чтобы заставить JVM ожидать отладчик. Прежде чем приступить к выполнению, он устанавливает пост отладчика на **5005** .

Обнаружение теста

Тестовое задание определяет, какие классы являются тестовыми, проверяя скомпилированные тестовые классы. По умолчанию он сканирует все файлы .class. Вы можете установить пользовательские включения / исключения, только те классы будут сканироваться. В зависимости от используемой среды тестирования (JUnit / TestNG) для определения класса теста используются разные критерии.

При использовании JUnit мы сканируем тестовые классы JUnit 3 и 4. Если какой-либо из следующих критериев соответствует, класс считается тестовым классом JUnit —

- Класс или суперкласс расширяет TestCase или GroovyTestCase
- Класс или суперкласс аннотируются @RunWith
- Класс или суперкласс содержат метод, аннотированный @Test
- При использовании TestNG мы сканируем методы, аннотированные @Test

Примечание . Абстрактные классы не выполняются. Gradle также сканирует дерево наследования в файлы JAR на тестовом пути к классам.

Если вы не хотите использовать обнаружение тестового класса, вы можете отключить его, установив для **ScanForTestClasses** значение false.

Тестовая группировка

JUnit и TestNG позволяют сложные группы методов тестирования. Для группировки тестовых классов и методов JUnit в JUnit 4.8 вводится понятие категорий. Тестовое задание позволяет указать категории JUnit, которые вы хотите включить и исключить.

Вы можете использовать следующий фрагмент кода в файле build.gradle для группировки методов тестирования.

```
test {
   useJUnit {
      includeCategories 'org.gradle.junit.CategoryA'
      excludeCategories 'org.gradle.junit.CategoryB'
```

```
   }
}
```

Включить и исключить отдельные тесты

Класс **Test** имеет метод **include** и **exclude** . Эти методы могут использоваться, чтобы указать, какие тесты должны фактически выполняться.

Запускайте только включенные тесты —

```
test {
   include '**my.package.name/*'
}
```

Пропустить исключенные тесты —

```
test {
   exclude '**my.package.name/*'
}
```

Пример файла **build.gradle,** как показано ниже, показывает различные параметры конфигурации.

```
apply plugin: 'java' // adds 'test' task

test {
   // enable TestNG support (default is JUnit)
   useTestNG()

   // set a system property for the test JVM(s)
   systemProperty 'some.prop', 'value'

   // explicitly include or exclude tests
   include 'org/foo/**'
   exclude 'org/boo/**'

   // show standard out and standard error of the test JVM(s) on the console
   testLogging.showStandardStreams = true

   // set heap size for the test JVM(s)
   minHeapSize = "128m"
   maxHeapSize = "512m"
```

```
   // set JVM arguments for the test JVM(s)
   jvmArgs '-XX:MaxPermSize=256m'

   // listen to events in the test execution lifecycle
   beforeTest {
      descriptor → logger.lifecycle("Running test: " + descriptor)
   }

   // listen to standard out and standard error of the test JVM(s)
   onOutput {
      descriptor, event → logger.lifecycle
         ("Test: " + descriptor + " produced standard out/err: "
         + event.message )
   }
}
```

Вы можете использовать следующий синтаксис команды для выполнения некоторого тестового задания.

```
gradle <someTestTask> --debug-jvm
```

Gradle — Multi-Project Build

Gradle может легко обрабатывать самые маленькие и самые крупные проекты. Небольшие проекты имеют один файл сборки и исходное дерево. Очень легко переварить и понять проект, который был разбит на более мелкие, взаимозависимые модули. Gradle прекрасно поддерживает этот сценарий, который состоит из нескольких проектов.

Структура для мультипроектной сборки

Такие сборки бывают разных форм и размеров, но у них есть некоторые общие характеристики —

- Файл **settings.gradle** в корневом или главном каталоге проекта.
- Файл **build.gradle** в корневом или главном каталоге.
- Дочерние каталоги, которые имеют свои собственные **файлы** сборки *** .gradle** (в некоторых многопроектных сборках могут отсутствовать сценарии сборки дочерних проектов).

Файл **settings.gradle** в корневом или главном каталоге проекта.

Файл **build.gradle** в корневом или главном каталоге.

Дочерние каталоги, которые имеют свои собственные **файлы** сборки *** .gradle** (в некоторых многопроектных сборках могут отсутствовать сценарии сборки дочерних проектов).

Для перечисления всех проектов в файле сборки вы можете использовать следующую команду.

```
C:\> gradle -q projects
```

Выход:

```
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

В отчете приведено описание каждого проекта, если он указан. Вы можете использовать следующую команду, чтобы указать описание. Вставьте его в файл **build.gradle** .

```
description = 'The shared API for the application'
```

Указание общей конфигурации сборки

В файле **build.gradle** в root_project общие конфигурации могут применяться ко всем проектам или только к подпроектам.

```
allprojects {
   group = 'com.example.gradle'
   version = '0.1.0'
}

subprojects {
   apply plugin: 'java'
   apply plugin: 'eclipse'
}
```

Это указывает общую группу **com.example.gradle** и версию **0.1.0** для всех проектов. Закрытие **подпроектов** применяет общие конфигурации для всех подпроектов, но не к корневому проекту, как закрытие всех проектов.

Конкретные конфигурации и зависимости проекта

Основные подпроекты пользовательского **интерфейса** и **утилиты** также могут иметь свой собственный файл **build.gradle** , если у них есть особые потребности, которые еще не применяются общей конфигурацией корневого проекта.

Например, проект пользовательского интерфейса обычно имеет зависимость от основного проекта. Таким образом, проекту пользовательского интерфейса нужен собственный файл **build.gradle,** чтобы указать эту зависимость.

```
dependencies {
   compile project(':core')
   compile 'log4j:log4j:1.2.17'
}
```

Зависимости проекта указываются с помощью метода проекта.

Gradle — Развертывание

Gradle предлагает несколько способов развертывания репозиториев артефактов сборки. При развертывании подписей для ваших артефактов в хранилище Maven вы также захотите подписать опубликованный файл POM.

Использование плагина Maven-publish

плагин **maven-publish** , предоставляемый Gradle по умолчанию. Используется для публикации скрипта Gradle. Посмотрите на следующий код.

```
apply plugin: 'java'
apply plugin: 'maven-publish'

publishing {
   publications {
      mavenJava(MavenPublication) {
         from components.java
      }
   }

   repositories {
      maven {
         url "$buildDir/repo"
      }
   }
}
```

Существует несколько опций публикации, когда применяется плагин **Java** и **maven-publish** . Посмотрите на следующий код, он развернет проект в удаленном хранилище.

```
apply plugin: 'groovy'
apply plugin: 'maven-publish'

group 'workshop'
version = '1.0.0'

publishing {
   publications {
      mavenJava(MavenPublication) {
         from components.java
      }
   }

   repositories {
      maven {
          default credentials for a nexus repository manager
         credentials {
            username 'admin'
            password 'admin123'
         }
         // url to the releases maven repository
            url "http://localhost:8081/nexus/content/repositories/releases/"
      }
   }
}
```

Преобразование проекта из Maven в Gradle

Для преобразования файлов Apache Maven **pom.xml в** файлы сборки Gradle существует специальная команда, если для этой задачи известны все используемые подключаемые модули Maven.

В этом разделе следующая конфигурация **pom.xml** maven будет преобразована в проект Gradle. Взгляните на это.

```
<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/xsd/maven-4.0.0.xsd">
```

```
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.example.app</groupId>
   <artifactId>example-app</artifactId>
   <packaging>jar</packaging>

   <version>1.0.0-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>

         <version>4.11</version>
         <scope>test</scope>
      </dependency>
   </dependencies>

</project>
```

Вы можете использовать следующую команду в командной строке, которая приводит к следующей конфигурации Gradle.

```
C:\> gradle init --type pom
```

Задача **init** зависит от задачи оболочки, поэтому создается оболочка Gradle.

Полученный файл **build.gradle** выглядит примерно так:

```
apply plugin: 'java'
apply plugin: 'maven'

group = 'com.example.app'
version = '1.0.0-SNAPSHOT'

description = """"""

sourceCompatibility = 1.5
targetCompatibility = 1.5

repositories {
   maven { url "http://repo.maven.apache.org/maven2" }
```

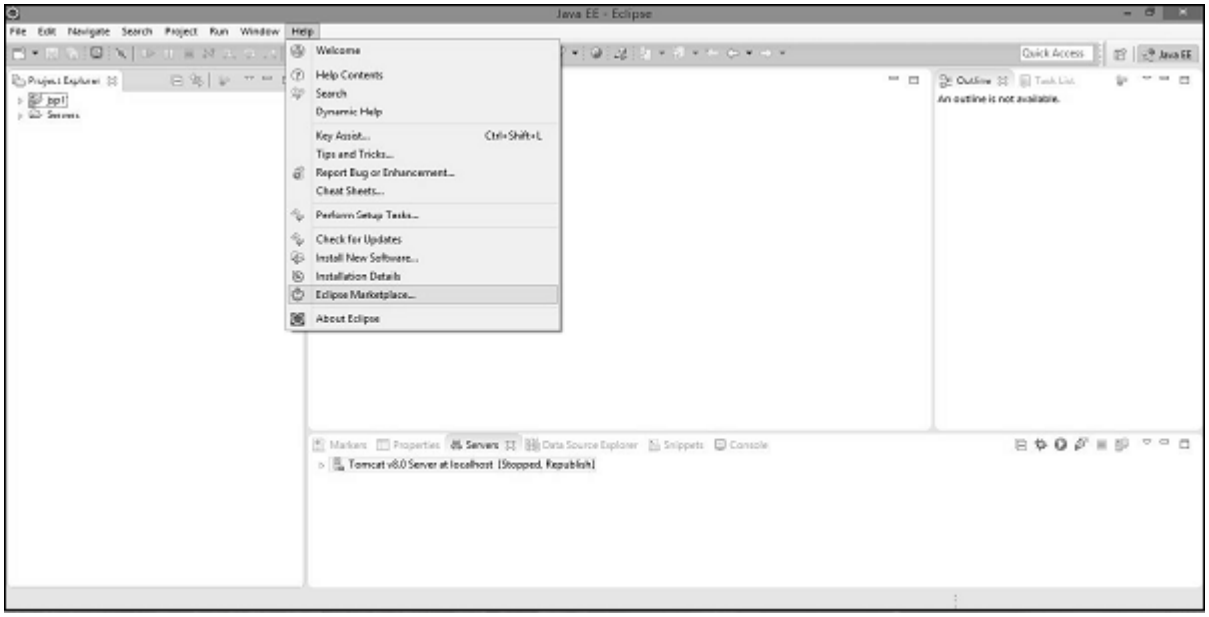
```
}


dependencies {
   testCompile group: 'junit', name: 'junit', version:'4.11'
}
```

Gradle — Eclipse Integration

В этой главе рассказывается об интеграции Eclipse и Gradle. Следуйте приведенным ниже инструкциям, чтобы добавить плагин Gradle для затмения.

Шаг 1 — Откройте Eclipse Marketplace

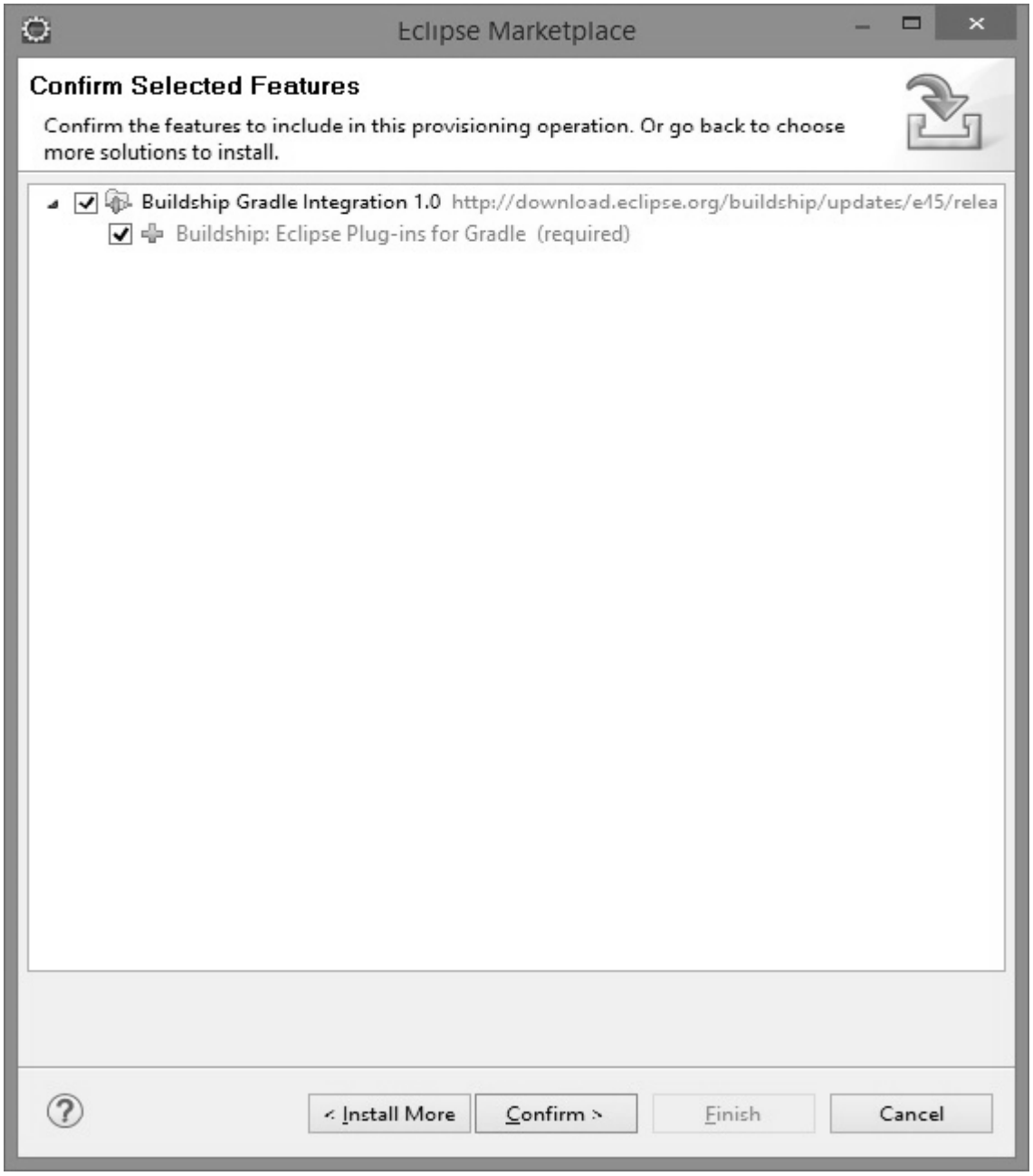
Прежде всего, откройте затмение, которое установлено в вашей системе. Перейти в помощь -> нажмите на EclipseMarketplace. Посмотрите на следующий скриншот.



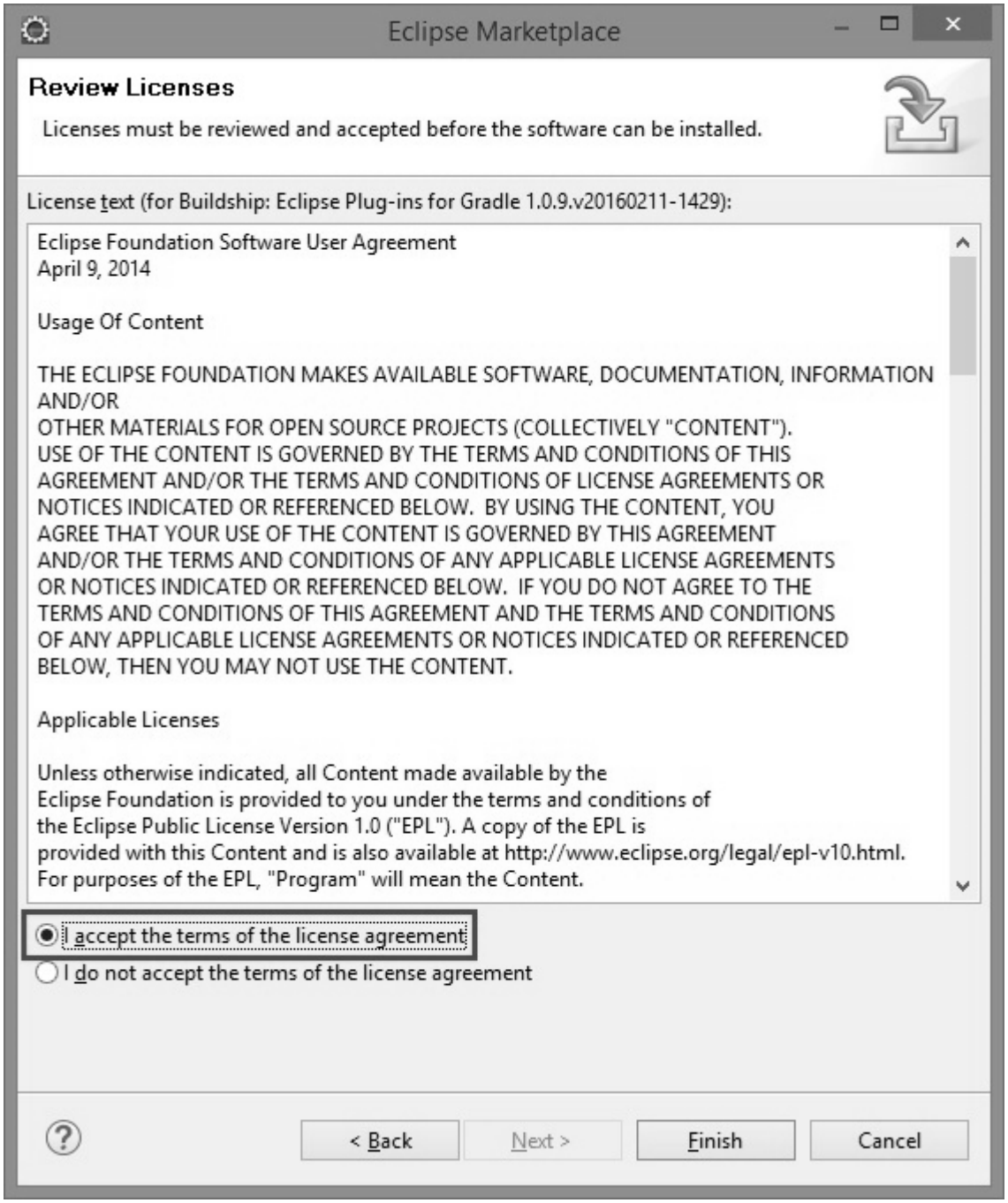
Шаг 2 — Установите плагин Buildship

После клика на Eclipse Marketplace вы увидите следующий скриншот. Здесь в левой части панели поиска типа **buildship** . Buildship — плагин интеграции Gradle. Когда вы найдете сборку на экране, нажмите «Установить» справа. Посмотрите на следующий скриншот.

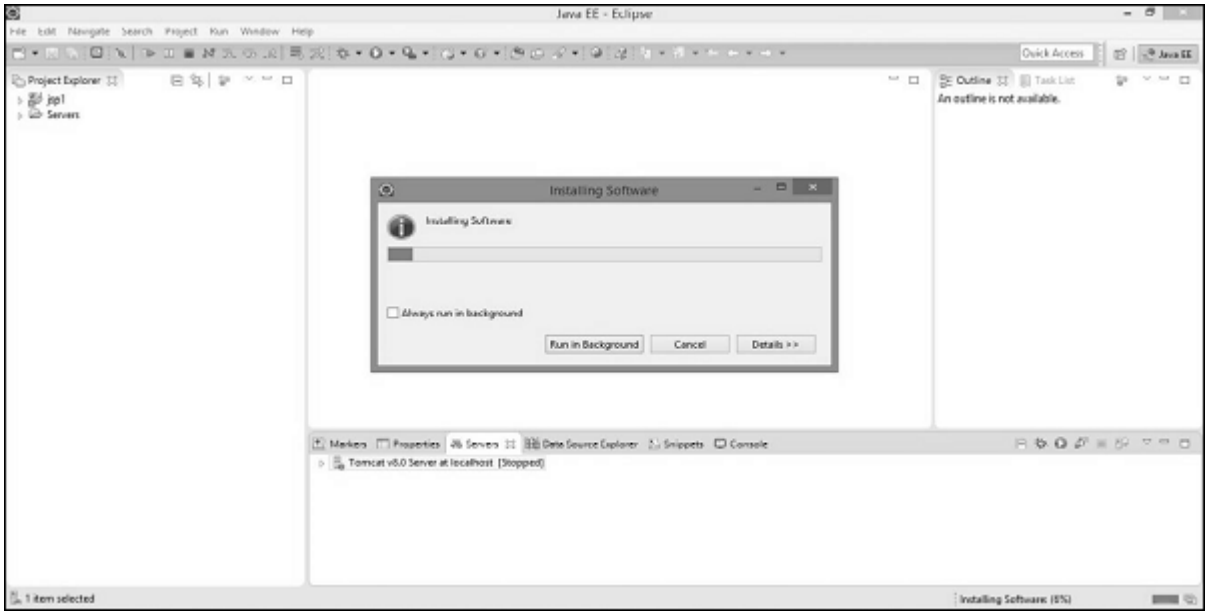
После этого вы найдете следующий снимок экрана, на котором вам нужно подтвердить установку программного обеспечения, нажав на кнопку подтверждения. Посмотрите на следующий скриншот.



После этого вам нужно нажать «Принять лицензионное соглашение» на следующем экране и нажать «Готово». Посмотрите на следующий скриншот.



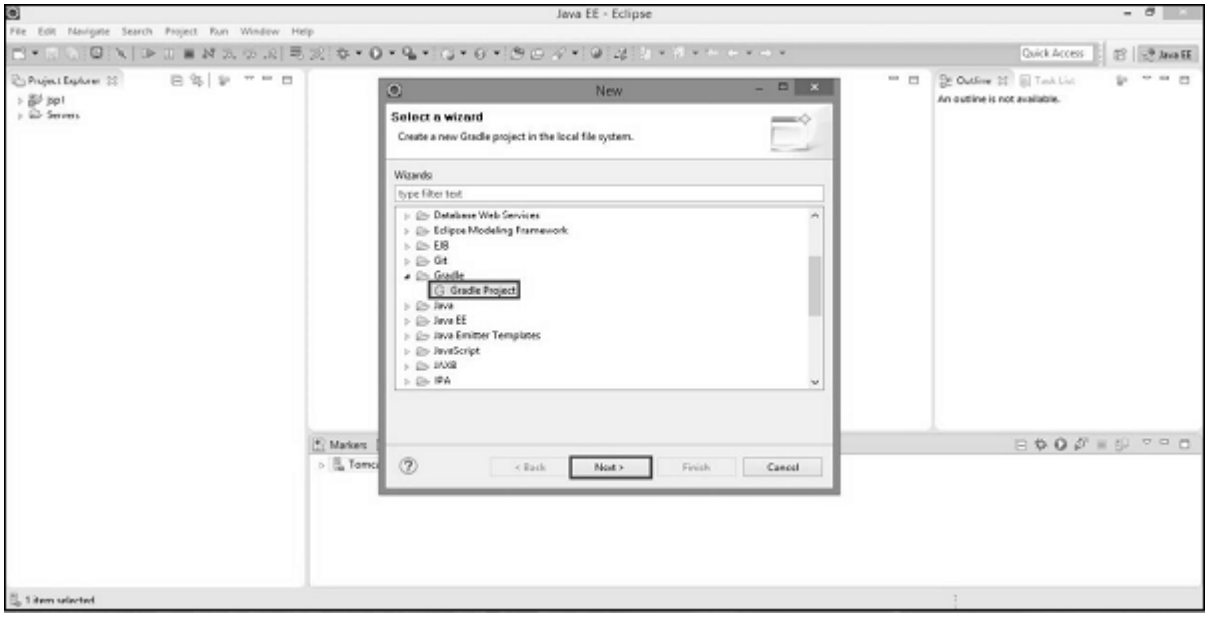
Это займет некоторое время для установки. Посмотрите на следующий скриншот.



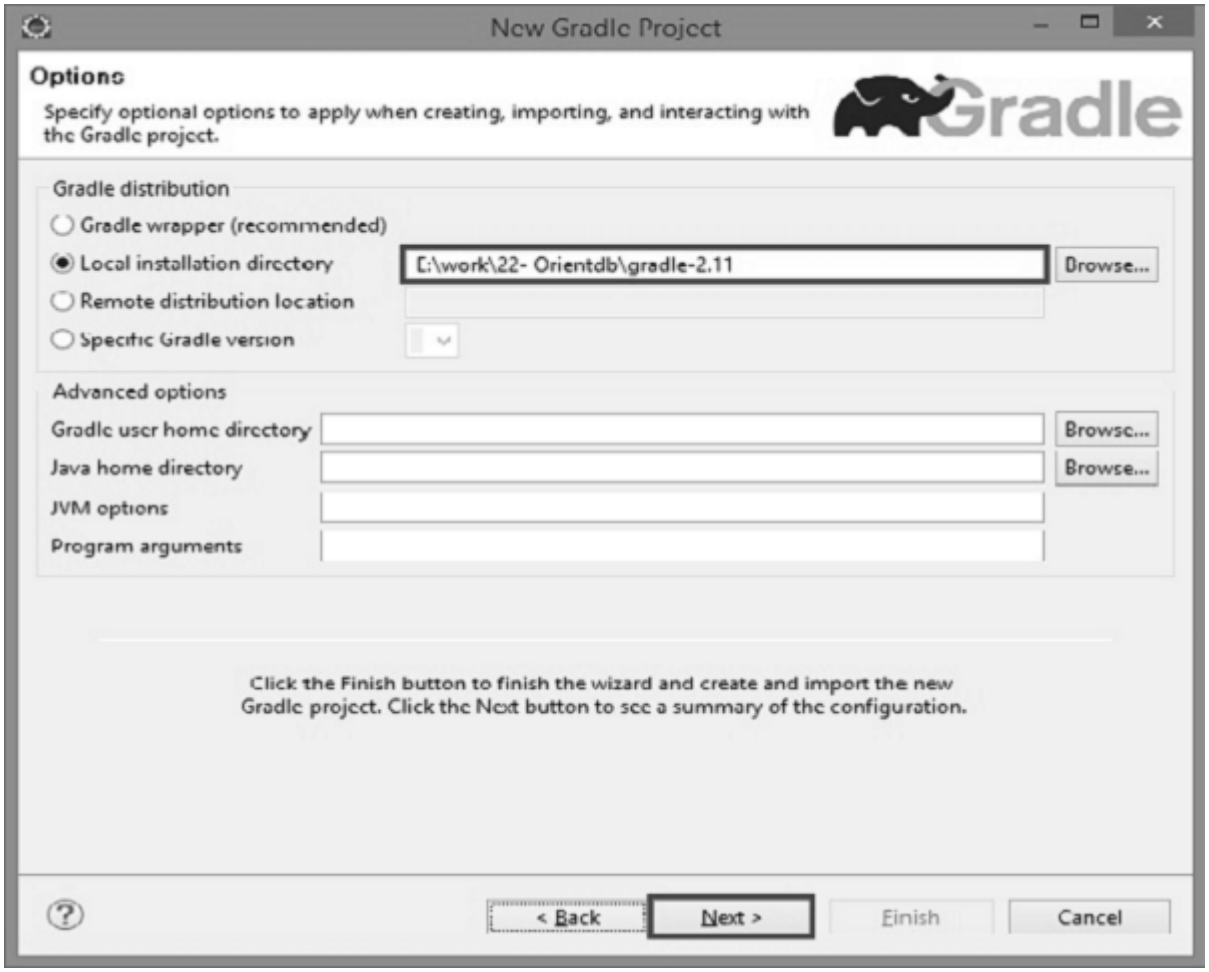
После этого он попросит перезапустить Eclipse. Там вы выберете **Да** .

Шаг 3 — Проверка плагина Gradle

При проверке мы создадим новый проект, следуя данной процедуре. В затмении перейдите в файл -> нажмите на новый -> нажмите на другие проекты. Там вы найдете следующий экран. Там выберите Gradle project и нажмите «Далее». Посмотрите на следующий снимок экрана.

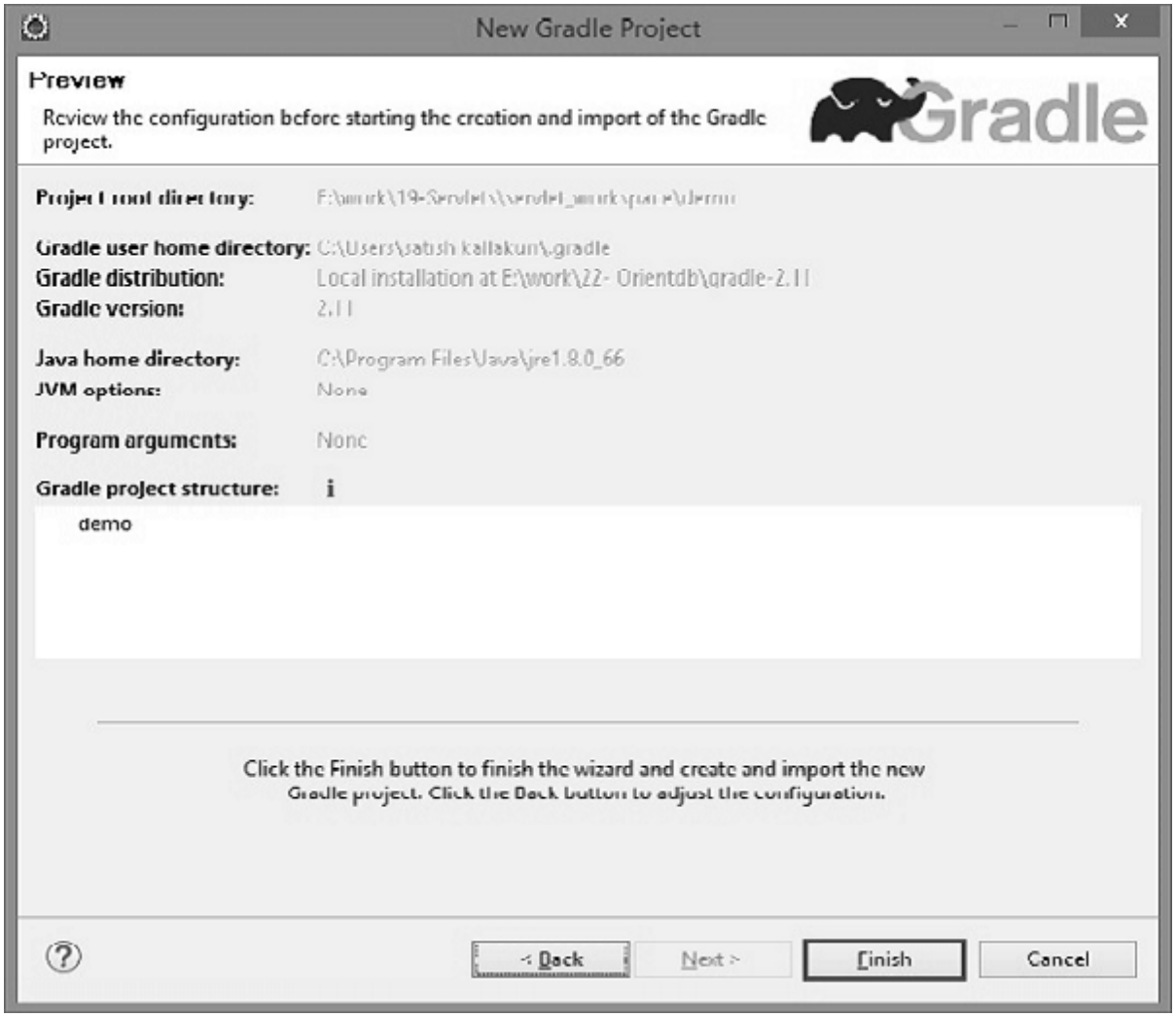


После нажатия следующей кнопки вы увидите следующий экран. Там вы укажете путь к домашней директории Gradle локальной файловой системы и нажмите кнопку «Далее». Посмотрите на следующий скриншот.



Взгляните на следующий скриншот, здесь вы дадите название проекту Gradle. В этом уроке мы используем **демопроект** и нажимаем кнопку « **Готово»** .

Взгляните на следующий скриншот, нам нужно подтвердить проект. Для этого у нас есть кнопка Готово на следующем экране.



Шаг 4 — Проверка структуры каталогов

После успешной установки плагина Gradle проверьте структуру каталогов демонстрационного проекта на наличие файлов и папок по умолчанию, как показано на следующем снимке экрана.

Тестирование программы, JUnit

JUnit — библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.

Библиотека **JUnit** была портирована на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CPPUnit), Flex (FlexUnit), JavaScript (JSUnit).

JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов. Опыт, полученный при работе с JUnit, важен в разработке концепций тестирования программного обеспечения.

Пример теста **JUnit**

```
import org.junit.Test;
import junit.framework.Assert;

public class MathTest {
    @Test
    public void testEquals() {
        Assert.assertEquals(4, 2 + 2);
        Assert.assertTrue(4 == 2 + 2);
    }

    @Test
    public void testNotEquals() {
        Assert.assertFalse(5 == 2 + 2);
    }
}
```

Необходимость использования JUnit

JUnit позволяет в любой момент быстро убедиться в работоспособности кода. Если программа не является совсем простой и включает множество классов и методов, то для её проверки может потребоваться значительное время. Естественно, что данный процесс лучше автоматизировать. Использование **JUnit** позволяет проверить код программы без значительных усилий и не занимает много времени.

Юнит тесты классов и функций являются своего рода документацией к тому, что ожидается в результате их выполнения. И не просто документацией, а документацией которая может автоматически проверять код на соответствие предъявленным функциям. Это удобно, и часто тесты разрабатывают как вместе, так и до реализации классов. Разработка через тестирование — крайне популярная технология создания серьезного программного обеспечения.

Виды тестирования и место JUnit тестирования в классификации

Тестирование программного обеспечение можно разделить на два вида:

- тестирование черного ящика;
- тестирование белого ящика.

Во время тестирования программы как черного ящика внутренняя структура приложения в расчет не принимается. Все, что имеет значение, это функциональность, которую приложение должно обеспечить. При тестировании программы как белого ящика во внимание принимается внутренняя структура, т.е. класс и методы. Кроме этого, тестирование можно разделить на четыре уровня:

- юнит тесты — тестирование отдельных участков кода;
- интеграционное тестирование — тестирование взаимодействия и совместной работы компонентов;
- системное тестирование — тестирование всей системы как целого;
- приемное тестирование — итоговое тестирование готовой системы на соответствие требованиям.

Юнит тестирование по определению является тестированием белого ящика.

Используется unit тестирование в двух вариантах - JUnit 3 и JUnit 4. Рассмотрим обе версии, так как в старых проектах до сих пор используется 3-я версия, которая поддерживает Java 1.4.

JUnit 3

Для создания теста следует наследовать тест-класс TestCase, переопределить методы setUp и tearDown при необходимости, ну и самое главное — разработать тестовые методы, наименование которых должно начинаться с аббривиатуры "test". При запуске теста сначала создается экземляр 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 вызывает исключение, тест считается провалившимся.

Примечание : тестовые методы должны быть public void, могут быть static.

Тесты состоят из выполнения некоторого кода и проверок. Проверки чаще всего выполняются с помощью класса **Assert** хотя иногда используют ключевое слово assert.

В качестве примера рассмотрим утилиту для работы со строками, включающую методы для проверки пустой строки и представления последовательности байт в виде 16-ричной строки:

```
public class JUnit3StringUtilsTest extends TestCase {
  private final Map toHexStringData = new HashMap();

  protected void setUp() throws Exception {
      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 });
      //...
  }

  protected void tearDown() throws Exception {
      toHexStringData.clear();
  }

  public void testToHexString() {
      for (Iterator iterator = toHexStringData.keySet().iterator();
                                                  iterator.hasNext();)
      {
          final String expected = (String)iterator.next();
          final byte[] testData = (byte[])toHexStringData.get(expected);
          final String actual = StringUtils.toHexString(testData);
          assertEquals(expected, actual);
      }
  }
  //...
}
```

Дополнительные возможности, TestSuite

JUnit 3 имеет несколько дополнительных возможностей. Например, можно группировать тесты. Для этого необходимо использовать класс **TestSuite**:

```
public class JUnit3StringUtilsTestSuite extends TestSuite {
  public JUnit3StringUtilsTestSuite() {
      addTestSuite(StringUtilsJUnit3Test.class);
      addTestSuite(OtherTest1.class);
      addTestSuite(OtherTest2.class);
  }
}
```

Можно исполнение теста повторить несколько раз. Для этого используется RepeatedTest :

```
public class JUnit3StringUtilsRepeatedTest extends RepeatedTest {
  public JUnit3StringUtilsRepeatedTest() {
      super(new JUnit3StringUtilsTest(), 100);
  }
}
```

Наследуя тест-класс от ExceptionTestCase, можно проверить код на выброс исключения :

```
public class JUnit3StringUtilsExceptionTest extends ExceptionTestCase {
  public JUnit3StringUtilsExceptionTest(final String name) {
      super(name, NullPointerException.class);
  }

  public void testToHexString() {
      StringUtils.toHexString(null);
  }
}
```

Как видно из примеров все довольно просто и ничего лишнего - минимум кода для JUnit тестирования.

JUnit 4

В JUnit 4 добавлена поддержка новых возможностей из Java 5.0; тесты могут быть объявлены с помощью аннотаций. При этом существует обратная совместимость с предыдущей версией фреймворка. Практически все рассмотренные выше примеры будут работать и в JUnit 4 за исключением RepeatedTest, который отсутствует в новой версии.

Какие внесены изменения появились в JUnit 4? Рассмотрим тот же пример, но уже с использованием новых возможностей :

```
public class JUnit4StringUtilsTest extends Assert {
  private final Map<String, byte[]> toHexStringData =
                                  new HashMap<String, byte[]>();

  @Before
  public static 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 static 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);
        }
    }
}
```

Что изменилось в JUnit 4 ?

- Для упрощения работы можно наследоваться от класса **Assert**, хотя это необязательно.
- Аннотация **@Before** обозначает методы, которые будут вызваны перед исполнением тестов. Методы должны быть *public void*. Здесь обычно размещаются предустановки для теста, в нашем случае это генерация тестовых данных (метод **setUpToHexStringData**).
- Можно использовать аннотацию **@BeforeClass**, которая обозначает методы, которые будут вызваны до создания экземпляра тест-класса; методы должны быть *public static void*. Данную аннотацию (метод) имеет смысл использовать для тестирования в случае, когда класс содержит несколько тестов, использующих различные предустановки, либо когда несколько тестов используют одни и те же данные, чтобы не тратить время на их создание для каждого теста.
- Аннотация **@After** обозначает методы, которые будут вызваны после выполнения тестов. Методы должны быть *public void*. Здесь размещаются операции освобождения ресурсов после теста; в нашем случае — очистка тестовых данных (метод **tearDownToHexStringData**).
- Аннотация **@AfterClass** связана по смыслу с **@BeforeClass**, но выполняет методы после тестирования класса. 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, в данной аннотации можно использовать два параметра, expected — задает ожидаемое исключение и timeout — задает время, по истечению которого тест считается провалившимся.

Примеры использования аннотаций с параметрами, JUnit Test :

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

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

Игнорирование выполнения теста, JUnit Ignore

Если один из тестов по какой-либо серьезной причине необходимо отключить, например, тест постоянно завершается с ошибкой. Исправление теста можно отложить до светлого будущего аннотированием **@Ignore**. Если поместить эту аннотацию на класс, то все тесты в этом классе будут отключены.

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

Правила тастирования, JUnit Rule

JUnit позволяет использовать определенные разработчиком правила до и после выполнения теста, которые расширяют функционал. Например, есть встроенные правила для задания таймаута для теста (Timeout), для задания ожидаемых исключений (ExpectedException), для работы с временными файлами(TemporaryFolder) и др.

Для объявления правила необходимо создать *public* не *static* поле типа производного от MethodRule и аннотировать его с помощью ключевого слова **Rule**.

```
public class JUnitOtherTest
{
    @Rule
    public final TemporaryFolder folder = new TemporaryFolder();

    @Rule
    public final Timeout timeout = new Timeout(1000);

    @Rule
    public final ExpectedException thrown = ExpectedException.none();

    @Ignore
    @Test
    public void anotherInfinity() {
        while (true);
    }

    @Test
    public void testFileWriting() throws IOException
    {
        final File log = folder.newFile("debug.log");
        final FileWriter logWriter = new FileWriter(log);
        logWriter.append("Hello, ");
        logWriter.append("World!!!");
        logWriter.flush();
        logWriter.close();
    }

    @Test
    public void testExpectedException() throws IOException
    {
        thrown.expect(NullPointerException.class);
        StringUtils.toHexString(null);
    }
}
```

Наборы тестов, JUnit Suite, SuiteClasses

Запуск теста может быть сконфигурирован с помощью аннотации **@RunWith**. Тестовые классы, которые содержат в себе тестовые методы, можно объединить в наборы тестов (Suite). Например, создано два класса тестирования объектов : TestFilter, TestConnect. Эти два тестовых класса можно объединить в один тестовый класс TestWidgets.java :

```
package com.objects;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses ({
    TestFilter.class,
    TestConnect.class
})

public class TestWidgets {}
```

Для настройки запускаемых тестов используется аннотация @SuiteClasses, в которую включены тестовые классы.

Аннотация Categories

Аннотация *Categories* позволяет объединить тесты в категории (группы). Для этого в тесте определяется категория *@Category*, после чего настраиваются запускаемые категории тестов в Suite. Это может выглядеть следующим образом:

```
public class JUnitStringUtilsCategoriesTest extends Assert
{
    //...

    @Category (Unit.class)
    @Test
    public void testIsEmpty() {
       //...
    }
    //...
}

@RunWith (Categories.class)
@Categories.IncludeCategory (Unit.class)
@Suite.SuiteClasses ({
    JUnitOtherTest.class,
        JUnitStringUtilsCategoriesTest.class
})
public class JUnitTestSuite {}
```

Аннотация, JUnit Parameterized

Аннотация *Parameterized* позволяет использовать параметризированные тесты. Для этого в тест-классе объявляется статический метод, возвращающий список данных, которые будут использованы в качестве аргументов конструктора класса.

```
@RunWith (Parameterized.class)
public class JUnitStringUtilsParameterizedTest extends Assert
{
    private final CharSequence testData;
    private final boolean expected;

    public JUnitStringUtilsParameterizedTest(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 },
        });
    }
}
```

Параметризирование метода : Theories.class, DataPoints, DataPoint, Theory

Аннотация **Theories** параметризирует тестовый метод, а не конструктор. Данные помечаются с помощью **@DataPoints** и **@DataPoint**, тестовый метод — с помощью **@Theory**. Тест, использующий этот функционал, может выглядеть примерно следующим образом :

```
@RunWith (Theories.class)
public class JUnitStringUtilsTheoryTest 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);
    }
}
```

Порядок выполнения тестов

Если необходимо выполнить тест в определенном порядке, то можно воспользоваться аннотацией @FixMethodOrder(MethodSorters.NAME_ASCENDING), определенной в JUnit 4.11. Например :

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class MyTest {
    @Test
        public void test01(){...}
    @Test
        public void test02(){...}
        ...
    @Test
        public void test09(){...}
}
```

В противном случае можно использовать следующие 2 подхода.

```
void test01();
void test02();
...
void test09();

@Test
public void testOrder1() { test1(); test3(); }

@Test(expected = Exception.class)
public void testOrder2() { test2(); test3(); test1(); }

@Test(expected = NullPointerException.class)
public void testOrder3() { test3(); test1(); test2(); }
@Test
public void testAllOrders() {
    for (Object[] sample: permute(1, 2, 3)) {
```

```
        for (Object index: sample) {
            switch (((Integer) index).intValue()) {
                case 1: test1(); break;
                case 2: test2(); break;
                case 3: test3(); break;
            }
        }
    }
}
```

Список основных аннотаций

Аннотация	Описание
@Test public void testMethod()	метод является тестовым
@Test(timeout=100) public void testMethod()	если время выполнения превысит параметр timeout, то тест будет завершен неудачно
@Test (expected = MyException.class) public void testMethod()	метод должен выбросить исключение принадлежащие к классу MyException, в противном случае тест будет завершен неудачно
@Ignore public void testMethod()	игнорировать тестовый метод
@BeforeClass public static void testMethod()	метод вызывающийся один раз для класса перед выполнением тестовых методов; здесь можно разместить инициализацию которую нужно выполнять только один раз, например, прочитать данные, которые будут использоваться в тестовых методах или создать соединение с базой данных
@AfterClass public static void testMethod()	метод вызывающийся один раз для класса после выполнения тестовых методов; здесь можно разместить деинициализацию которую нужно выполнять только один раз, например, закрыть соединение с базой данных или удалить данные, которые больше не нужны
@Before public static void beforeMethod()	метод, вызывающийся перед каждым тестовым методом в тестовом классе; здесь можно выполнить необходимую инициализацию, например, выставить начальные параметры
@After public static void afterMethod()	метод, вызывающийся после каждого тестового метода в тестовом классе; здесь можно выполнить необходимую деинициализацию, например, удалить данные, которые больше не нужны

Список типов проверок Asserts

Тип проверки	Описание
fail() fail(String message)	прерывание теста с ошибкой, т.е. тест будет неудачным
assertTrue(boolean condition) assertTrue(java.lang.String message, boolean condition)	проверка на равенство условия condition значению true
assertFalse(boolean condition) assertFalse(String message, boolean condition)	проверка на равенство условия condition значению false
assertEquals(<тип> expected, <тип> actual) assertEquals(String message, <тип> expected, <тип> actual)	проверка на равенство; <тип> — это Object, int, double и т.д.
assertArrayEquals(byte[] expecteds, byte[] actuals) assertArrayEquals(String message, <тип>[] expecteds, <тип>[] actuals)	проверка массивов на равенство; аналогично assertEquals; <тип> — это Object, int, double и т.д.
assertNotNull(Object object) assertNotNull(String message, Object object)	проверка, что Object не null
assertNull(Object object) assertNull(String message, Object object)	проверка, что Object null
assertSame(Object expected, Object actual) assertSame(String message, Object expected, Object actual)	проверка на равенство двух объектов expected и actual, т.е. один и тот же объект

Пример JUnit тестирования

Для демонстрации основных возможностей JUnit, используем примитивный java класс FuncMath, который имеет два метода - нахождение факториала неотрицательного числа и суммы двух чисел. Кроме того, в экземпляре класса будет находится счетчик вызовов методов.

```
public class FuncMath
{
    int calls;

    public int getCalls() {
        return calls;
    }
    public long factorial(int number) {
        calls++;

        if (number < 0)
            throw new IllegalArgumentException();

        long result = 1;
        if (number > 1) {
            for (int i = 1; i < = number; i++)
```

```
                result = result * i;
        }

        return result;
    }

    public long plus(int num1, int num2) {
        calls++;
        return num1 + num2;
    }
}
```

Иногда для выполнения каждого тестового сценария необходим некоторый контекст, например, заранее созданные экземпляры классов. А после выполнения нужно освободить зарезервированные ресурсы. В этом случае используют аннтоации @Before и @After. Метод, помеченный @Before будет выполняться перед каждым тестовым случаем, а метод, помеченный @After - после каждого тестового случая. Если же инициализацию и освобождение ресурсов нужно сделать всего один раз - соответственно до и после всех тестов - то используют пару аннотаций @BeforeClass и @AfterClass.

Тестовый класс с несколькими сценариями будет иметь следующий вид :

```
import org.junit.Test;
import org.junit.After;
import org.junit.Before;
import org.junit.Assert;
import org.junit.AfterClass;
import org.junit.BeforeClass;

public class JUnit_funcMath extends Assert
{
    private FuncMath math;

    @Before
    public void init() {
        math = new FuncMath();
    }
    @After
    public void tearDown() {
        math = null;
    }
    @Test
    public void calls() {
        assertEquals("math.getCalls() != 0", 0, dao.getConnection());

        math.factorial(1);
        assertEquals(1, math.getCalls());

        math.factorial(1);
        assertEquals(2, math.getCalls());
    }

    @Test
    public void factorial() {
        assertTrue(math.factorial(0) == 1);
        assertTrue(math.factorial(1) == 1);
        assertTrue(math.factorial(5) == 120);
    }

    @Test(expected = IllegalArgumentException.class)
    public void factorialNegative() {
        math.factorial(-1);
    }

    @Ignore
```

```
    @Test
    public void todo() {
        assertTrue(math.plus(1, 1) == 3);
    }
}
```

Метод calls тестирует правильность счетчика вызовов. Метод factorial проверяет правильность вычисления факториала для некоторых стандартных значений. Метод factorialNegative проверяет, что для отрицательных значений факотриала будет брошен IllegalArgumentException. Метод todo будет проигнорирован.

В заключении следует отметить, что в статье представлены не все возможности использования JUnit. Но как видно из приведенных примеров, фреймворк достаточно прост в использовании, дополнительных возможностей немного, но есть возможность расширения с помощью правил и запускалок

JUnit и фреймворк Mockito

Разработчикам программного обеспечения на разных этапах своей деятельности приходится сталкиваться с тремя стратегиями тестирования
: *функциональным*, *интеграционным* и ***модульным*** тестированием. Все они используются для тестирования приложений разными способами, и каждая из стратегий имеет определенную цель. К сожалению, ни одна из стратегий не даёт стопроцентной гарантии обнаружения всех имеющихся ошибок. И даже комбинация всех трёх стратегий не может дать такой гарантии. Но их сочетание позволяет существенно снизить количество ошибок и убедить разработчика, что приложение функционирует согласно предъявленным требованиям.

Функциональное тестирование

Проведение функционального тестирования, как правило, связано с созданием специальной группы специалистов, занимающихся тестированием. На этом этапе приложение развертывается в действующем окружении и проверяется его соответствие ТЗ (Техническому Заданию) и предъявленным функциональным требованиям. Команда тестеровщиков использует комплекс автоматизированных и ручных тестов.

Автоматизировать процесс функционального тестирования можно, если приложение включает API (Application Programming Interface) - интерфейс прикладного программирования, на котором оно построено. Однако наличие интерфейса в приложении (desktop, web) существенно снижает возможности полной автоматизации данного процесса.

Интеграционное тестирование

Стратегия интеграционного тестирования основывается на проверке прикладного кода в окружении, близком к фактическому окружению, но не являющимся им. Главная цель данной стратегии – убедиться в правильности взаимодействия кода с внешними ресурсами и взаимодействия различных технологий в приложении между собой.

В интеграционном тестировании не требуется использовать фиктивные данные, как при модульном тестировании. Вместо этого в интеграционных тестах часто используются базы данных, находящиеся в памяти, которые легко можно создавать и уничтожать во время выполнения тестов. База данных в памяти – это самая настоящая база данных, что дает возможность проверить правильность работы сущностей JPA. Но все же эта база данных не совсем настоящая – она лишь имитирует настоящую базу данных для целей интеграционного тестирования.

Модульное тестирование

Целью *модульного тестирования* является проверка работы прикладной логики всего приложения или отдельных его частей при разных исходных данных, и анализ правильности получаемых результатов. Несмотря на то, что цель **модульного тестирования** выглядит простой и понятной, реализация этого типа тестирования может оказаться очень сложным и запутанным делом, особенно при наличии «старого» кода. Основные приемы проведения модульного тестирования опираются на следующие базовые принципы :

- внешние ресурсы не используются, т.е. недопустимо подключение к базам данных, веб-службам и т.п.;
- каждый класс имеет свой тест;
- тестируются только общедоступные методы или интерфейсы, а внутренний код тестируется за счет изменения входных данных;
- для получения данных, требуемых тестируемой логике, должны создаваться фиктивные зависимости.

При проведении модульного тестирования для создания фиктивных классов-зависимостей можно использовать простой, но мощный фреймворк **Mockito** совместно с JUnit.

Фреймворк Mockito

Фреймворк *Mockito* предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit тестов. *Mockito* можно скачать с сайта https://code.google.com/p/mockito, либо определить в зависимостях (dependencies) в maven проекте :

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
```

Наибольшее распространение получили следующие возможности *Mockito* :

- создание заглушек для классов и интерфейсов;
- проверка вызыва метода и значений передаваемых методу параметров;
- использование концепции «частичной заглушки», при которой заглушка создается на класс с определением поведения, требуемое для некоторых методов класса;
- подключение к реальному классу «шпиона» *spy* для контроля вызова методов.

Синтаксис создания заглушки Mockito

Чтобы создать *Mockito* объект можно использовать либо аннотацию @Mock, либо метод *mock*. В следующем примере в двух разных классах (Test_Mockito1, Test_Mockito2) разными способами создаются объекты mcalc для имитации интерфейса калькулятора ICalculator.

```
import org.mockito.Mock;
import org.mockito.Mockito;

import com.example ICalculator;

public class Test_Mockito1
{
    @Mock
    ICalculator mcalc;
}

-------------------------------------------------------------

public class Test_Mockito2
{
    ICalculator mcalc = Mockito.mock(ICalculator.class);
}
```

Если использовать статический импорт *Mockito*, то синтаксис будет иметь следующий вид :

```
import static org.mockito.Mockito.*;

import com.example ICalculator;

public class Test_Mockito
{
    ICalculator mcalc = mock(ICalculator.class);
}
```

*ПРИМЕЧАНИЕ : помните, что методы **mock** объекта возвращают значения по умолчанию : false для boolean, 0 для int, пустые коллекции, null для остальных объектов.*

Методы Mockito в примерах

Для рассмотрения методов фреймворка **Mockito** будем использовать в качестве тестового класса калькулятор, реализующий интерфейс ICalculator. В следующем коде представлены листинги интерфейса ICalculator и класса Calculator.

Листинги тестового класса и интерфейса

```
public interface ICalculator
{
    public double add      (double d1, double d2);
    public double subtract (double d1, double d2);
    public double multiply (double d1, double d2);
    public double divide   (double d1, double d2);
}
//-------------------------------------------------
public class Calculator
{
    ICalculator icalc;

    public Calculator(ICalculator icalc) {
        this.icalc = icalc;
    }
    public double add(double d1, double d2) {
        return icalc.add(d1, d2);
    }
    public double subtract(double d1, double d2) {
        return icalc.subtract(d1, d2);
    }
    public double multiply(double d1, double d2) {
        return icalc.multiply(d1, d2);
    }
    public double divide(double d1, double d2) {
        return icalc.divide(d1, d2);
    }
    public double double15() {
        return 15.0;
    }
}
```

Как видно из листингов все методы калькулятора (add, subtract, multiply, divide), за исключением одного, возвращают не вычисленные значения, а результаты выполнения данных методов в объекте, реализующим интерфейс ICalculator, который в наших тестах будет представлять заглушка *mcalc*. Последний метод *double15()* должен вернуть реальное значение.

1. Определение поведения - when(mock).thenReturn(value)

Этот метод позволяет определить возвращаемое значение при вызове метода *mock* с заданными параметрами. Если будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернётся последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Таким образом, чтобы метод всегда возвращал одно и то же значение, седует просто определить одно условие.

```
@RunWith(MockitoJUnitRunner.class)
public class Test_Mockito
{
    @Mock
    ICalculator mcalc;

    // используем аанотацию @InjectMocks для создания mock объекта
    @InjectMocks
    Calculator calc = new Calculator(mcalc);

    @Test
    public void testCalcAdd()
```

```
    {
        // определяем поведение калькулятора для операции сложения
        when(calc.add(10.0, 20.0)).thenReturn(30.0);

        // проверяем действие сложения
        assertEquals(calc.add(10, 20), 30.0, 0);
        // проверяем выполнение действия
        verify(mcalc).add(10.0, 20.0);

        // определение поведения с использованием doReturn
        doReturn(15.0).when(mcalc).add(10.0, 5.0);

        // проверяем действие сложения
        assertEquals(calc.add(10.0, 5.0), 15.0, 0);
        verify(mcalc).add(10.0, 5.0);
    }
}
```

Метод *verify* позволяет проверить, была ли выполнена проверка с определенными параметрами. Если проверка не выполнялась или выполнялась с другими параметрами, то *verify* вызовет исключение.

Для определения поведения *mock* в тесте была использована также следующая конструкция :

```
doReturn(value).when(mock).method(params)
```

2. Подсчет количества вызовов - atLeast, atLeastOnce, atMost, times, never

Для проверки количества вызовов определенных методов *Mockito* предоставляет следующие методы :

- **atLeast (int min)** - не меньше min вызовов;
- **atLeastOnce ()** - хотя бы один вызов;
- **atMost (int max)** - не более max вызовов;
- **times (int cnt)** - cnt вызовов;
- **never ()** - вызовов не было;

Следующий тест демонстрирует контроль количества вызовов метода *subtract* с разными параметрами. Для этого сначала определяется поведение *mock* (при определенных параметрах выдавать соответствующие результаты), и выполняются проверки с использованием *assertEquals*. После этого выполняется проверка количества вызовов *mock* с определенными параметрами. Две последние проверки не выполняются - «закомментированы». Если снять комментарий хотя бы с одной из них, то метод *verify*, вызовет исключение. Комментарий к этим проверкам описывает причину вызова методом исключения.

```
@Test
public void testCallMethod()
{
    // определяем поведение (результаты)
    when(mcalc.subtract(15.0, 25.0)).thenReturn(10.0);
    when(mcalc.subtract(35.0, 25.0)).thenReturn(-10.0);

    // вызов метода subtract и проверка результата
    assertEquals (calc.subtract(15.0, 25.0), 10, 0);
    assertEquals (calc.subtract(15.0, 25.0), 10, 0);

    assertEquals (calc.subtract(35.0, 25.0),-10, 0);

    // проверка вызова методов
    verify(mcalc, atLeastOnce()).subtract(35.0, 25.0);
```

```
    verify(mcalc, atLeast   (2)).subtract(15.0, 25.0);

    // проверка - был ли метод вызван 2 раза?
    verify(mcalc, times(2)).subtract(15.0, 25.0);
    // проверка - метод не был вызван ни разу
    verify(mcalc, never()).divide(10.0,20.0);

    /* Если снять комментарий со следующей проверки, то
     * ожидается exception, поскольку метод "subtract"
     * с параметрами (35.0, 25.0) был вызван 1 раз
     */
    // verify(mcalc, atLeast (2)).subtract(35.0, 25.0);

    /* Если снять комментарий со следующей проверки, то
     * ожидается exception, поскольку метод "subtract"
     * с параметрами (15.0, 25.0) был вызван 2 раза, а
     * ожидался всего один вызов
     */
    // verify(mcalc, atMost (1)).subtract(15.0, 25.0);
}
```

3. Обработка исключений - when(mock).thenThrow()

Mockito позволяет вызвать исключение при определенных условиях. Для этого необходимо использовать следующий синтаксис кода :

```
// создаем исключение
RuntimeException exception = new RuntimeException ("Division by zero");
// определение поведения для вызова исключения
doThrow(exception).when(mock).divide(5.0, 0));
```

В представленном коде создали исключение RuntimeException. После этого определили условия вызова исключения - вызов метода деления на 0. В следующем тесте выполняется проверка метода divide. При делении на 0 вызывается исключение.

```
@Test
public void testDevide()
{
    when(mcalc.divide(15.0, 3)).thenReturn(5.0);

    assertEquals(calc.divide(15.0, 3), 5.0, 0);
    // проверка вызова метода
    verify(mcalc).divide(15.0, 3);

    // создаем исключение
    RuntimeException exception = new RuntimeException ("Division by zero");
    // определяем поведение
    doThrow(exception).when(mcalc).divide(15.0, 0);

    assertEquals(calc.divide(15.0, 0), 0.0, 0);
    verify(mcalc).divide(15.0, 0);
}
```

4. Использование интерфейса org.mockito.stubbing.Answer<T>

Иногда описание поведения *mock* объекта требует определенной проверки с усложнением логики. В этом случае можно использовать интерфейс **Answer<T>**, который позволяет реализовать заглушки методов со сложным поведением. В следующем тесте *testThenAnswer* при вызове метода сложения с определенными параметрами *mcalc.add(11.0, 12.0)* будет вызван метод *answer*, который подготовит ответ. Параметр *InvocationOnMock* позволяет получить информацию о вызываемом методе и параметрах.

```
// метод обработки ответа
private Answer<Double> answer = new Answer<Double>() {
    @Override
    public Double answer(InvocationOnMock invocation) throws Throwable
    {
        // получение объекта mock
        Object mock = invocation.getMock();
        System.out.println ("mock object : " + mock.toString());

        // аргументы метода, переданные mock
        Object[] args = invocation.getArguments();
        double d1 = (double) args[0];
        double d2 = (double) args[1];
        double d3 = d1 + d2;
        System.out.println ("" + d1 + " + " + d2);

        return d3;
    }
};
@Test
public void testThenAnswer()
{
    // определение поведения mock для метода с параметрами
    when(mcalc.add(11.0, 12.0)).thenAnswer(answer);
    assertEquals(calc.add(11.0,12.0), 23.0, 0);
}
```

5. Использование шпиона spy на реальных объектах

Mockito позволяет подключать к реальным объектам «шпиона» **spy**, контролировать возвращаемые методами значения и отслеживать количество вызовов методов. В следующем тесте создадим шпиона *scalc*, который подключим к реальному калькулятору и будем вызывать метод *double15()*. Необходимо отметить, что метод реального объекта *double15* должен вернуть значение 15. Однако *Mockito* позволяет переопределить значение и согласно вновь назначенному условию новое значение должно быть 23.

```
@Test
public void testSpy()
{
    Calculator scalc = spy(new Calculator());
    when(scalc.double15()).thenReturn(23.0);

    // вызов метода реального класса
    scalc.double15();
    // проверка вызова метода
    verify(scalc).double15();

    // проверка возвращаемого методом значения
    assertEquals(23.0, scalc.double15(), 0);
    // проверка вызова метода не менее 2-х раз
    verify(scalc, atLeast(2)).double15();
}
```

В результате выполнения теста видим, что метод возвращает значение 23. Таким образом, фреймворк **Mockito** в сочетании с JUnit можно использовать для тестов реального класса. При этом, можно проверить, вызывался ли метод, сколько раз и с какими параметрами. Кроме этого, можно создавать заглушки только для некоторых методов. Это позволяет проверить поведение одних методов, используя заглушки для других.

6. Проверка вызова метода с задержкой, timeout

Фреймворк *Mockito* позволяет выполнить проверку вызова определенного метода в течение заданного в **timeout** времени. Задержка времени определяется в милисекундах.

```
@Test
public void testTimout()
{
    // определение результирующего значения mock для метода
    when(mcalc.add(11.0, 12.0)).thenReturn(23.0);
    // проверка значения
    assertEquals(calc.add(11.0,12.0), 23.0, 0);

    // проверка вызова метода в течение 10 мс
    verify(mcalc, timeout(100)).add(11.0, 12.0);
}
```

7. Использование в тестах java классов

В следующем тесте при создании *mock* объектов используются java классы Iterator и Comparable. После этого определяются условия проверок и выполняются тесты.

```
@Test
public void testJavaClasses()
{
    // создание объекта mock
    Iterator<String> mis = mock(Iterator.class);
    // формирование ответов
    when(mis.next()).thenReturn("Привет").thenReturn("Mockito");
    // формируем строку из ответов
    String result = mis.next() + ", " + mis.next();
    // проверяем
    assertEquals("Привет, Mockito", result);

    Comparable<String> mcs = mock(Comparable.class);
    when(mcs.compareTo("Mockito")).thenReturn(1);
    assertEquals(1, mcs.compareTo("Mockito"));

    Comparable<Integer> mci = mock(Comparable.class);
    when(mci.compareTo(anyInt())).thenReturn(1);
    assertEquals(1, mci.compareTo(5));
}
```

Краткое руководство по BDDMockito

- Testing Mockito

1. Обзор

- Термин BDD был впервые введен Dan North - в 2006 году

BDD поощряет написание тестов на естественном, понятном человеку языке, ориентированном на поведение приложения

Он определяет четко структурированный способ написания тестов в следующих трех разделах (Arrange, Act, Assert):

- *given* некоторые предварительные условия (Arrange)
- *when* действие происходит (закон)
- *then* проверить вывод (Утвердить)
- Библиотека Mockito поставляется с классом *BDDMockito* , который представляет дружественные к BDD API. ** Этот API позволяет нам использовать более дружественный к BDD подход, организуя наши тесты с помощью *given ()* и делая утверждения с помощью *then ()* .

В этой статье мы собираемся объяснить, как настроить наши тесты Mockito на основе BDD. Мы также поговорим о различиях между API *Mockito* и *BDDMockito* , чтобы в конечном итоге сосредоточиться на API *BDDMockito* .

2. Настроить

2.1. Зависимости Maven

- BDD-версия Mockito является частью библиотеки *mockito-core* ** , чтобы начать, нам просто нужно включить артефакт:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
</dependency>
```

Для получения последней версии Mockito, пожалуйста, проверьте Maven Центральная .

2.2. Импорт

Наши тесты могут стать более читабельными, если мы включим следующий статический импорт:

```
import static org.mockito.BDDMockito.** ;
```

Обратите внимание, что *BDDMockito* расширяет *Mockito* , поэтому мы не пропустим ни одной функции, предоставляемой традиционным *Mockito* API.

3. Мокито против BDDMockito

Традиционный макет в Mockito выполняется с помощью *when (obj)* . *then* ** *()* на шаге Arrange.

Позже, взаимодействие с нашим макетом может быть проверено с помощью *verify ()* на шаге Assert.

- *BDDMockito* предоставляет псевдонимы BDD для различных методов *Mockito* , поэтому мы можем написать наш шаг Arrange, используя *given* (вместо *when*), также мы можем написать наш шаг Assert, используя *then* (вместо *verify*).

Давайте рассмотрим пример тестового тела с использованием традиционного Mockito:

```
when(phoneBookRepository.contains(momContactName))
  .thenReturn(false);
```

```
phoneBookService.register(momContactName, momPhoneNumber);

verify(phoneBookRepository)
  .insert(momContactName, momPhoneNumber);
```
Давайте посмотрим, как это можно сравнить с *BDDMockito* :

```
given(phoneBookRepository.contains(momContactName))
  .willReturn(false);

phoneBookService.register(momContactName, momPhoneNumber);

then(phoneBookRepository)
  .should()
  .insert(momContactName, momPhoneNumber);
```
4. Дразнить с *BDDMockito*

Давайте попробуем протестировать *PhoneBookService* , где нам нужно будет посмеяться над *PhoneBookRepository:*

```
public class PhoneBookService {
    private PhoneBookRepository phoneBookRepository;

    public void register(String name, String phone) {
        if(!name.isEmpty() && !phone.isEmpty()
          && !phoneBookRepository.contains(name)) {
            phoneBookRepository.insert(name, phone);
        }
    }

    public String search(String name) {
        if(!name.isEmpty() && phoneBookRepository.contains(name)) {
            return phoneBookRepository.getPhoneNumberByContactName(name);
        }
        return null;
    }
}
```
BDDMockito as *Mockito* позволяет нам возвращать значение, которое может быть фиксированным или динамическим. Это также позволило бы нам создать исключение:

4.1. Возврат фиксированного значения

Используя *BDDMockito,* мы могли бы легко настроить Mockito для возврата фиксированного результата всякий раз, когда вызывается наш целевой метод ложного объекта:

```
given(phoneBookRepository.contains(momContactName))
  .willReturn(false);

phoneBookService.register(xContactName, "");

then(phoneBookRepository)
  .should(never())
  .insert(momContactName, momPhoneNumber);
```
4.2. Возврат динамического значения

BDDMockito позволяет нам предоставить более сложный способ возврата значений. Мы могли бы вернуть динамический результат на основе ввода:

```
given(phoneBookRepository.contains(momContactName))
  .willReturn(true);
given(phoneBookRepository.getPhoneNumberByContactName(momContactName))
  .will((InvocationOnMock invocation) ->
    invocation.getArgument(0).equals(momContactName)
      ? momPhoneNumber
      : null);
phoneBookService.search(momContactName);
then(phoneBookRepository)
```

```
  .should()
  .getPhoneNumberByContactName(momContactName);
```

4.3. Бросать исключение

Сказать, что Mockito сгенерирует исключение, довольно просто:

```
given(phoneBookRepository.contains(xContactName))
  .willReturn(false);
willThrow(new RuntimeException())
  .given(phoneBookRepository)
  .insert(any(String.class), eq(tooLongPhoneNumber));

try {
    phoneBookService.register(xContactName, tooLongPhoneNumber);
    fail("Should throw exception");
} catch (RuntimeException ex) { }

then(phoneBookRepository)
  .should(never())
  .insert(momContactName, tooLongPhoneNumber);
```

Обратите внимание, как мы поменялись местами *given* и *will* ** , что является обязательным в случае, если мы высмеиваем метод, который не имеет возвращаемого значения.

Также обратите внимание, что мы использовали сопоставители аргументов, такие как (*any* , *eq*), чтобы обеспечить более общий способ насмешки, основанный на критериях, а не на фиксированном значении.

5. Заключение

В этом кратком руководстве мы обсудили, как BDDMockito пытается создать сходство BDD с нашими тестами Mockito, и обсудили некоторые различия между *Mockito* и *BDDMockito* .

Как всегда, исходный код можно найти over на GitHub - в тестовом пакете *com.baeldung.bddmockito*

Junit 5. Basic

Annotations

JUnit Jupiter supports the following annotations for configuring tests and extending the framework.

Unless otherwise stated, all core annotations are located in the `org.junit.jupiter.api` package in the `junit-jupiter-api` module.

Annotation	Description
`@Test`	Denotes that a method is a test method. Unlike JUnit 4's `@Test` annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are *inherited* unless they are *overridden*.
`@ParameterizedTest`	Denotes that a method is a parameterized test. Such methods are *inherited* unless they are *overridden*.
`@RepeatedTest`	Denotes that a method is a test template for a repeated test. Such methods are *inherited* unless they are *overridden*.
`@TestFactory`	Denotes that a method is a test factory for dynamic tests. Such methods are *inherited* unless they are *overridden*.
`@TestTemplate`	Denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such methods are *inherited* unless they are *overridden*.
`@TestMethodOrder`	Used to configure the test method execution order for the annotated test class; similar to JUnit 4's `@FixMethodOrder`. Such annotations are *inherited*.
`@TestInstance`	Used to configure the test instance lifecycle for the annotated test class. Such annotations are *inherited*.
`@DisplayName`	Declares a custom display name for the test class or test method. Such annotations are not *inherited*.
`@DisplayNameGeneration`	Declares a custom display name generator for the test class. Such annotations are *inherited*.
`@BeforeEach`	Denotes that the annotated method should be executed *before* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@Before`. Such methods are *inherited* unless they are *overridden*.
`@AfterEach`	Denotes that the annotated method should be executed *after* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@After`. Such methods are *inherited* unless they are *overridden*.

Annotation	Description
`@BeforeAll`	Denotes that the annotated method should be executed *before* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@BeforeClass`. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be `static` (unless the "per-class" test instance lifecycle is used).
`@AfterAll`	Denotes that the annotated method should be executed *after* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@AfterClass`. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be `static` (unless the "per-class" test instance lifecycle is used).
`@Nested`	Denotes that the annotated class is a non-static nested test class. `@BeforeAll` and `@AfterAll` methods cannot be used directly in a `@Nested` test class unless the "per-class" test instance lifecycle is used. Such annotations are not *inherited*.
`@Tag`	Used to declare tags for filtering tests, either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are *inherited* at the class level but not at the method level.
`@Disabled`	Used to disable a test class or test method; analogous to JUnit 4's `@Ignore`. Such annotations are not *inherited*.
`@Timeout`	Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are *inherited*.
`@ExtendWith`	Used to register extensions declaratively. Such annotations are *inherited*.
`@RegisterExtension`	Used to register extensions programmatically via fields. Such fields are *inherited* unless they are *shadowed*.
`@TempDir`	Used to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method; located in the `org.junit.jupiter.api.io` package.

Some annotations may currently be *experimental*. Consult the table in Experimental APIs for details.

Meta-Annotations and Composed Annotations

JUnit Jupiter annotations can be used as *meta-annotations*. That means that you can define your own *composed annotation* that will automatically *inherit* the semantics of its meta-annotations.

For example, instead of copying and pasting `@Tag("fast")` throughout your code base (see Tagging and Filtering), you can create a custom *composed annotation* named `@Fast` as follows. `@Fast` can then be used as a drop-in replacement for `@Tag("fast")`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
public @interface Fast {
}
```

The following `@Test` method demonstrates usage of the `@Fast` annotation.

```
@Fast
@Test
void myFastTest() {
    // ...
}
```

You can even take that one step further by introducing a custom `@FastTest` annotation that can be used as a drop-in replacement for `@Tag("fast")` *and* `@Test`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
@Test
public @interface FastTest {
}
```

JUnit automatically recognizes the following as a `@Test` method that is tagged with "fast".

```
@FastTest
void myFastTest() {
    // ...
}
```

Переход с JUnit 4 на JUnit 5

- Testing JUnit 5

1. Обзор

В этой статье мы увидим, как мы можем перейти с JUnit 4 на последнюю версию JUnit 5 - с обзором различий между двумя версиями библиотеки.

Общие рекомендации по использованию JUnit 5 см. В нашей ссылке на статью:/junit-5[здесь].

2. JUnit 5 Преимущества

Давайте начнем с предыдущей версии - JUnit 4 имеет некоторые явные ограничения:

- Вся структура содержалась в одной библиотеке jar. Целый

Библиотека должна быть импортирована, даже если требуется. **В JUnit 5 мы получаем больше детализации и можем импортировать только то, что является необходимым** ** Один исполнитель тестов может одновременно выполнять тесты только в JUnit 4 (например,

SpringJUnit4ClassRunner или *Parameterized*). **JUnit 5 позволяет несколько бегунов работать одновременно** ** JUnit 4 никогда не выходил за пределы Java 7, пропуская множество функций

из Java 8. **JUnit 5 хорошо использует функции Java 8**

Идея JUnit 5 заключалась в том, чтобы полностью переписать JUnit 4, чтобы устранить большинство из этих недостатков.

3. Отличия

JUnit 4 был разделен на модули, которые составляют JUnit 5:

- **JUnit Platform -** этот модуль охватывает все платформы расширений, которые мы

может быть заинтересован в выполнении теста, обнаружения и отчетности JUnit Vintage - ** этот модуль обеспечивает обратную совместимость с JUnit

4 или даже JUnit 3

3.1. Аннотации

JUnit 5 содержит важные изменения в своих аннотациях. **Наиболее важным является то, что мы больше не можем использовать аннотацию *@ Test* для указания ожиданий.**

Параметр *expected* в JUnit 4:

```
@Test(expected = Exception.class)
public void shouldRaiseAnException() throws Exception {
    //...
}
```

Теперь мы можем использовать метод *assertThrows* :

```
public void shouldRaiseAnException() throws Exception {
    Assertions.assertThrows(Exception.class, () -> {
        //...
    });
}
```

Атрибут *timeout* в JUnit 4:

```
@Test(timeout = 1)
public void shouldFailBecauseTimeout() throws InterruptedException {
    Thread.sleep(10);
}
```
Теперь метод *assertTimeout* в JUnit 5:

```
@Test
public void shouldFailBecauseTimeout() throws InterruptedException {
    Assertions.assertTimeout(Duration.ofMillis(1), () -> Thread.sleep(10));
}
```
Другие аннотации, которые были изменены в JUnit 5:

- *@ Before* аннотация переименована в *@ BeforeEach*
- *@ After* аннотация переименована в *@ AfterEach*
- *@ BeforeClass* аннотация переименована в *@ BeforeAll*
- *@ AfterClass* аннотация переименована в *@ AfterAll*
- *@ Ignore* аннотация переименована в *@ Disabled*

3.2. Утверждения

Теперь мы можем написать сообщения с утверждениями в лямбда-выражениях в JUnit 5, что позволяет при отложенной оценке пропускать сложное построение сообщения до тех пор, пока это не понадобится:

```
@Test
public void shouldFailBecauseTheNumbersAreNotEqual__lazyEvaluation() {
    Assertions.assertTrue(
      2 == 3,
      () -> "Numbers " + 2 + " and " + 3 + " are not equal!");
}
```
Мы также можем сгруппировать утверждения в JUnit 5:

```
@Test
public void shouldAssertAllTheGroup() {
    List<Integer> list = Arrays.asList(1, 2, 4);
    Assertions.assertAll("List is not incremental",
        () -> Assertions.assertEquals(list.get(0).intValue(), 1),
        () -> Assertions.assertEquals(list.get(1).intValue(), 2),
        () -> Assertions.assertEquals(list.get(2).intValue(), 3));
}
```
3.3. Предположения

Новый класс *Assumptions* теперь находится в *org.junit.jupiter.api.Assumptions* . JUnit 5 полностью поддерживает существующие методы предположений в JUnit 4, а также добавляет набор новых методов, позволяющих запускать некоторые утверждения только в определенных сценариях:

```
@Test
public void whenEnvironmentIsWeb__thenUrlsShouldStartWithHttp() {
    assumingThat("WEB".equals(System.getenv("ENV")),
      () -> {
          assertTrue("http".startsWith(address));
      });
}
```
3.4. Пометка и фильтрация

В JUnit 4 мы могли группировать тесты, используя аннотацию *@ Category* .

В JUnit 5 аннотация *@ Category* заменяется аннотацией *@ Tag* :

```
@Tag("annotations")
@Tag("junit5")
@RunWith(JUnitPlatform.class)
public class AnnotationTestExampleTest {
   /** ...** /}
```

Мы можем включить/исключить определенные теги, используя *maven-surefire-plugin* :

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <properties>
                    <includeTags>junit5</includeTags>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

3.5. Новые аннотации для запуска тестов

@ RunWith использовался для интеграции тестового контекста с другими платформами или для изменения общего потока выполнения в тестовых примерах в JUnit 4.

С помощью JUnit 5 теперь мы можем использовать аннотацию *@ ExtendWith* для обеспечения аналогичной функциональности.

В качестве примера, чтобы использовать функции Spring в JUnit 4:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
  {"/app-config.xml", "/test-data-access-config.xml"})
public class SpringExtensionTest {
   /** ...** /}
```

Теперь в JUnit 5 это простое расширение:

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(
  { "/app-config.xml", "/test-data-access-config.xml" })
public class SpringExtensionTest {
   /** ...** /}
```

3.6. Новые аннотации правил испытаний

В JUnit 4 аннотации *@ Rule* и *@ ClassRule* использовались для добавления специальных функций в тесты.

В JUnit 5. мы можем воспроизвести ту же логику, используя аннотацию *@ ExtendWith* .

Например, скажем, у нас есть пользовательское правило в JUnit 4 для записи трассировок журнала до и после теста:

```
public class TraceUnitTestRule implements TestRule {

    @Override
    public Statement apply(Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
               //Before and after an evaluation tracing here
                ...
            }
        };
    }
}
```

И мы реализуем это в тестовом наборе:

```
@Rule
public TraceUnitTestRule traceRuleTests = new TraceUnitTestRule();
```

В JUnit 5 мы можем написать то же самое гораздо более интуитивно понятным способом:

```
public class TraceUnitExtension implements AfterEachCallback, BeforeEachCallback {

    @Override
    public void beforeEach(TestExtensionContext context) throws Exception {
        //...
    }

    @Override
    public void afterEach(TestExtensionContext context) throws Exception {
        //...
    }
}
```

Используя интерфейсы *AfterEachCallback* и *BeforeEachCallback* в JUnit 5, доступные в пакете *org.junit.jupiter.api.extension,* мы легко реализуем это правило в комплекте тестов:

```
@RunWith(JUnitPlatform.class)
@ExtendWith(TraceUnitExtension.class)
public class RuleExampleTest {

    @Test
    public void whenTracingTests() {
        /** ...** /    }
}
```

3.7. Юнит 5 Винтаж

JUnit Vintage помогает в миграции тестов JUnit, выполняя тесты JUnit 3 или JUnit 4 в контексте JUnit 5.

Мы можем использовать его, импортировав JUnit Vintage Engine:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>${junit5.vintage.version}</version>
    <scope>test</scope>
</dependency>
```

4. Заключение

Как мы видели в этой статье, JUnit 5 - это модульная и современная версия платформы JUnit 4. Мы ввели основные различия между этими двумя версиями и намекнули, как перейти с одной на другую

Руководство по расширению JUnit 5

- Testing JUnit 5

1. Обзор

В этой статье мы рассмотрим модель расширения в библиотеке тестирования JUnit 5. Как следует из названия, **цель расширений Junit 5 состоит в том, чтобы расширить поведение тестовых классов или методов** , и они могут быть повторно использованы для нескольких тестов.

До 5 июня версия библиотеки JUnit 4 использовала два типа компонентов для расширения теста: исполнители тестов и правила. Для сравнения, JUnit 5 упрощает механизм расширения, вводя единую концепцию: API *Extension* .

2. Модель расширения JUnit 5

Расширения JUnit 5 относятся к определенному событию при выполнении теста, называемому точкой расширения. Когда достигается определенная фаза жизненного цикла, механизм JUnit вызывает зарегистрированные расширения.

Можно использовать пять основных типов точек расширения:

- постобработка тестового экземпляра
- условное выполнение теста
- обратные вызовы жизненного цикла
- разрешение параметра
- Обработка исключений

Мы рассмотрим каждый из них более подробно в следующих разделах.

3. Зависимости Maven

Во-первых, давайте добавим зависимости проекта, которые нам понадобятся для наших примеров.

Нам понадобится основная библиотека JUnit 5: *junit-jupiter-engine* :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.1.0</version>
    <scope>test</scope>
</dependency>
```

Также давайте добавим две вспомогательные библиотеки для использования в наших примерах:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.8.2</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.196</version>
</dependency>
```

Последние версии junit-jupiter-engine , https://search.maven.org/classic/#search % 7Cga% 7C1% 7Ca% 3A% 22h2% 22% 20AND% 20g% 3A% 22com.h2database% 22[h2]и https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A % 22log4j-core% 22% 20AND% 20g% 3A% 22org.apache.logging.log4j% 22[log4j-core]можно загрузить из Maven Central.

4. Создание расширений JUnit 5

Чтобы создать расширение JUnit 5, нам нужно определить класс, который реализует один или несколько интерфейсов, соответствующих точкам расширения JUnit 5. Все эти интерфейсы расширяют основной интерфейс *Extension* , который является только интерфейсом маркера.

4.1. *TestInstancePostProcessor* Extension

Расширение этого типа выполняется после того, как был создан экземпляр теста. Интерфейс для реализации - *TestInstancePostProcessor* , у которого есть метод *postProcessTestInstance ()* для переопределения.

Типичным вариантом использования этого расширения является внедрение зависимостей в экземпляр. Например, давайте создадим расширение, которое создает экземпляр объекта *logger* , а затем вызывает метод *setLogger ()* в тестовом экземпляре:

```
public class LoggingExtension implements TestInstancePostProcessor {

    @Override
    public void postProcessTestInstance(Object testInstance,
      ExtensionContext context) throws Exception {
        Logger logger = LogManager.getLogger(testInstance.getClass());
        testInstance.getClass()
          .getMethod("setLogger", Logger.class)
          .invoke(testInstance, logger);
    }
}
```

Как видно выше, метод *postProcessTestInstance ()* обеспечивает доступ к экземпляру теста и вызывает метод *setLogger ()* класса теста, используя механизм отражения.

4.2. Условное выполнение теста

JUnit 5 предоставляет тип расширения, который может контролировать, следует ли запускать тест. Это определяется реализацией интерфейса *ExecutionCondition* .

Давайте создадим класс *EnvironmentExtension* , который реализует этот интерфейс и переопределяет метод *evaluateExecutionCondition ()* .

Метод проверяет, равняется ли свойство, представляющее имя текущей среды, *«qa»* , и отключает тест в этом случае:

```
public class EnvironmentExtension implements ExecutionCondition {

    @Override
    public ConditionEvaluationResult evaluateExecutionCondition(
      ExtensionContext context) {

        Properties props = new Properties();
        props.load(EnvironmentExtension.class
          .getResourceAsStream("application.properties"));
        String env = props.getProperty("env");
        if ("qa".equalsIgnoreCase(env)) {
            return ConditionEvaluationResult
              .disabled("Test disabled on QA environment");
        }

        return ConditionEvaluationResult.enabled(
          "Test enabled on QA environment");
    }
}
```

В результате тесты, которые регистрируют это расширение, не будут выполняться в среде *«qa»* .

- Если мы не хотим, чтобы условие было проверено, мы можем деактивировать его, установив ключ конфигурации *junit.conditions.deactivate* ** в шаблон, соответствующий условию.

Это может быть достигнуто путем запуска JVM со свойством *-Djunit.conditions.deactivate = <pattern>* или путем добавления параметра конфигурации в *LauncherDiscoveryRequest* :

```
public class TestLauncher {
    public static void main(String[]args) {
        LauncherDiscoveryRequest request
          = LauncherDiscoveryRequestBuilder.request()
          .selectors(selectClass("com.baeldung.EmployeesTest"))
          .configurationParameter(
            "junit.conditions.deactivate",
            "com.baeldung.extensions.** ")
          .build();
```

```
        TestPlan plan = LauncherFactory.create().discover(request);
        Launcher launcher = LauncherFactory.create();
        SummaryGeneratingListener summaryGeneratingListener
          = new SummaryGeneratingListener();
        launcher.execute(
          request,
          new TestExecutionListener[]{ summaryGeneratingListener });

        System.out.println(summaryGeneratingListener.getSummary());
    }
}
```

4.3. Обратные вызовы жизненного цикла

Этот набор расширений связан с событиями в жизненном цикле теста и может быть определен путем реализации следующих интерфейсов:

- *BeforeAllCallback* и *AfterAllCallback* - выполняется до и после

все методы испытаний выполнены ** *BeforeEachCallBack* и *AfterEachCallback* - выполняется до и

после каждого метода испытаний ** *BeforeTestExecutionCallback* и *AfterTestExecutionCallback* -

выполняется непосредственно перед и сразу после метода испытаний

Если тест также определяет методы его жизненного цикла, порядок выполнения такой:

, *BeforeAllCallback*

, *BeforeAll*

, *BeforeEach Callback*

, *BeforeEach*

, *BeforeTestExecutionCallback*

, *Тестовое задание*

, *AfterTestExecutionCallback*

, *AfterEach*

, *AfterEachCallback*

, В конце концов

, *AfterAllCallback*

В нашем примере давайте определим класс, который реализует некоторые из этих интерфейсов и управляет поведением теста, который обращается к базе данных с помощью JDBC.

Сначала давайте создадим простую сущность *Employee* :

```
public class Employee {

    private long id;
    private String firstName;
   //constructors, getters, setters
}
```

Нам также понадобится служебный класс, который создает *Connection* на основе файла *.properties* :

```
public class JdbcConnectionUtil {

    private static Connection con;

    public static Connection getConnection()
      throws IOException, ClassNotFoundException, SQLException{
        if (con == null) {
           //create connection
            return con;
        }
        return con;
    }
}
```

Наконец, давайте добавим простой *DAO* на основе JDBC, который манипулирует записями *Employee* :

```
public class EmployeeJdbcDao {
    private Connection con;

    public EmployeeJdbcDao(Connection con) {
        this.con = con;
    }

    public void createTable() throws SQLException {
       //create employees table
    }

    public void add(Employee emp) throws SQLException {
      //add employee record
    }

    public List<Employee> findAll() throws SQLException {
      //query all employee records
    }
}
```

- Давайте создадим наше расширение, которое реализует некоторые интерфейсы жизненного цикла: **

```
public class EmployeeDatabaseSetupExtension implements
  BeforeAllCallback, AfterAllCallback, BeforeEachCallback, AfterEachCallback {
    //...
}
```

Каждый из этих интерфейсов содержит метод, который мы должны переопределить.

Для интерфейса *BeforeAllCallback* мы переопределим метод *beforeAll ()* и добавим логику для создания нашей таблицы *employees* перед выполнением любого тестового метода:

```
private EmployeeJdbcDao employeeDao = new EmployeeJdbcDao();

@Override
public void beforeAll(ExtensionContext context) throws SQLException {
    employeeDao.createTable();
}
```

Далее мы будем использовать *BeforeEachCallback* и *AfterEachCallback* , чтобы обернуть каждый тестовый метод в транзакции. Цель этого - откатить любые изменения в базе данных, выполненные в методе test, чтобы следующий тест выполнялся на чистой базе данных.

В методе *beforeEach ()* мы создадим точку сохранения, которая будет использоваться для отката состояния базы данных:

```
private Connection con = JdbcConnectionUtil.getConnection();
private Savepoint savepoint;

@Override
public void beforeEach(ExtensionContext context) throws SQLException {
    con.setAutoCommit(false);
    savepoint = con.setSavepoint("before");
}
```

Затем в методе *afterEach ()* мы откатим изменения в базе данных, сделанные во время выполнения тестового метода:

```
@Override
public void afterEach(ExtensionContext context) throws SQLException {
    con.rollback(savepoint);
}
```

Чтобы закрыть соединение, мы будем использовать метод *afterAll ()* , который выполняется после завершения всех тестов:

```
@Override
public void afterAll(ExtensionContext context) throws SQLException {
    if (con != null) {
        con.close();
    }
}
```

4.4. Разрешение параметра

Если конструктор или метод теста получает параметр, он должен быть разрешен во время выполнения *ParameterResolver* .

Давайте определим наш собственный *ParameterResolver* , который разрешает параметры типа *EmployeeJdbcDao* :

```
public class EmployeeDaoParameterResolver implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
      ExtensionContext extensionContext) throws ParameterResolutionException {
        return parameterContext.getParameter().getType()
          .equals(EmployeeJdbcDao.class);
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext,
      ExtensionContext extensionContext) throws ParameterResolutionException {
        return new EmployeeJdbcDao();
    }
}
```

Наш распознаватель реализует интерфейс *ParameterResolver* и переопределяет методы *supportsParameter ()* и *resolveParameter ()* . Первый из них проверяет тип параметра, а второй определяет логику для получения экземпляра параметра.

4.5. Обработка исключений

И последнее, но не менее важное: интерфейс *TestExecutionExceptionHandler* можно использовать для определения поведения теста при обнаружении определенных типов исключений.

Например, мы можем создать расширение, которое будет регистрировать и игнорировать все исключения типа *FileNotFoundException* , при этом перебрасывая любой другой тип:

```
public class IgnoreFileNotFoundExceptionExtension
  implements TestExecutionExceptionHandler {

    Logger logger = LogManager
      .getLogger(IgnoreFileNotFoundExceptionExtension.class);

    @Override
    public void handleTestExecutionException(ExtensionContext context,
      Throwable throwable) throws Throwable {

        if (throwable instanceof FileNotFoundException) {
            logger.error("File not found:" + throwable.getMessage());
            return;
        }
        throw throwable;
```

```
    }
}
```

5. Регистрация расширений

Теперь, когда мы определили наши тестовые расширения, нам нужно зарегистрировать их с помощью теста JUnit 5. Чтобы достичь этого, мы можем использовать аннотацию *@ ExtendWith* .

Аннотацию можно добавить несколько раз в тест или получить список расширений в качестве параметра:

```
@ExtendWith({ EnvironmentExtension.class,
  EmployeeDatabaseSetupExtension.class, EmployeeDaoParameterResolver.class })
@ExtendWith(LoggingExtension.class)
@ExtendWith(IgnoreFileNotFoundExceptionExtension.class)
public class EmployeesTest {
    private EmployeeJdbcDao employeeDao;
    private Logger logger;

    public EmployeesTest(EmployeeJdbcDao employeeDao) {
        this.employeeDao = employeeDao;
    }

    @Test
    public void whenAddEmployee__thenGetEmployee() throws SQLException {
        Employee emp = new Employee(1, "john");
        employeeDao.add(emp);
        assertEquals(1, employeeDao.findAll().size());
    }

    @Test
    public void whenGetEmployees__thenEmptyList() throws SQLException {
        assertEquals(0, employeeDao.findAll().size());
    }

    public void setLogger(Logger logger) {
        this.logger = logger;
    }
}
```

Мы видим, что в нашем тестовом классе есть конструктор с параметром *EmployeeJdbcDao* , который будет разрешен путем расширения расширения *EmployeeDaoParameterResolver* .

При добавлении *EnvironmentExtension* наш тест будет выполняться только в среде, отличной от *"qa"*

В нашем тесте также будет создана таблица *employees* и каждый метод будет заключен в транзакцию путем добавления *EmployeeDatabaseSetupExtension* .

Даже если сначала выполняется тест *whenAddEmployee* thenGetEmploee () __, который добавляет одну запись в таблицу, второй тест найдет 0 записей в таблице.

Экземпляр регистратора будет добавлен в наш класс с помощью *LoggingExtension* .

Наконец, наш тестовый класс будет игнорировать все экземпляры *FileNotFoundException* , так как он добавляет соответствующее расширение.

5.1. Автоматическая регистрация продления

Если мы хотим зарегистрировать расширение для всех тестов в нашем приложении, мы можем сделать это, добавив полное имя в файл */META-INF/services/org.junit.jupiter.api.extension.Extension* :

```
com.baeldung.extensions.LoggingExtension
```

Чтобы этот механизм был включен, нам также нужно установить для ключа конфигурации *junit.extensions.autodetection.enabled* значение true. Это можно сделать, запустив JVM со свойством *– Djunit.extensions.autodetection.enabled = true* или добавив параметр конфигурации в *LauncherDiscoveryRequest* :

```
LauncherDiscoveryRequest request
```

```
  = LauncherDiscoveryRequestBuilder.request()
  .selectors(selectClass("com.baeldung.EmployeesTest"))
  .configurationParameter("junit.extensions.autodetection.enabled", "true")
.build();
```

6. Заключение

В этом руководстве мы показали, как мы можем использовать модель расширений JUnit 5 для создания пользовательских тестовых расширений

Mockito и JUnit 5 - Использование

- Testing Mockito JUnit 5

1. Вступление

В этой быстрой статье мы покажем **, как интегрировать Mockito с моделью расширения JUnit 5** . Чтобы узнать больше о модели расширения JUnit 5, взгляните на этот article .

Сначала мы покажем, как создать расширение, которое автоматически создает фиктивные объекты для любого атрибута класса или параметра метода, помеченного *@ Mock* .

Затем мы будем использовать наше расширение Mockito в тестовом классе JUnit 5.

2. Зависимости Maven

2.1. Обязательные зависимости

Давайте добавим зависимости JUnit 5 (jupiter) и *mockito* в наш *pom.xml* :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
    <scope>test</scope>
</dependency>
```

Обратите внимание, что *_unit-jupiter-engine - это основная библиотека JUnit 5, а junit-platform-launcher_* используется с плагином Maven и средством запуска IDE.

2.2. Плагин Surefire

Давайте также сконфигурируем плагин Maven Surefire для запуска наших тестовых классов с помощью новой платформы запуска JUnit:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <dependencies>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-surefire-provider</artifactId>
            <version>1.0.1</version>
        </dependency>
    </dependencies>
</plugin>
```

2.3. Зависимости JUnit 4 IDE

Чтобы наши тесты были совместимы с JUnit4 (vintage), для IDE, которые еще не поддерживают JUnit 5, давайте включим следующие зависимости:

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.2.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.vintage</groupId>
```

```
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```
Кроме того, мы должны рассмотреть возможность аннотирования всех наших тестовых классов с помощью *@ RunWith (JUnitPlatform.class)*

Последние версии *junit-jupiter-engine* , https://search.maven.org/classic/#search % 7Cga% 7C1% 7Cjunit% 20vintage% 20engine[junit-vintage-engine], *junit-platform-launcher* , и *mockito-core* можно загрузить из Maven Central.

3. Расширение Mockito

Mockito обеспечивает реализацию расширений JUnit5 в библиотеке - *https://search.maven.org/search? Q = a: mockito-junit-jupiter[mockito-junit-jupiter]*

Мы включим эту зависимость в наш *pom.xml:*

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>2.23.0</version>
    <scope>test</scope>
</dependency>
```
4. Создание тестового класса

Давайте создадим наш тестовый класс и добавим к нему расширение Mockito:

```
@ExtendWith(MockitoExtension.class)
@RunWith(JUnitPlatform.class)
public class UserServiceUnitTest {

    UserService userService;

...//}
```
Мы можем использовать аннотацию *@ Mock* , чтобы добавить макет для переменной экземпляра, которую мы можем использовать в любом месте тестового класса:

```
@Mock UserRepository userRepository;
```
Также мы можем ввести фиктивные объекты в параметры метода:

```
@BeforeEach
void init(@Mock SettingRepository settingRepository) {
    userService = new DefaultUserService(userRepository, settingRepository, mailClient);

    Mockito.lenient().when(settingRepository.getUserMinAge()).thenReturn(10);

    when(settingRepository.getUserNameMinLength()).thenReturn(4);


Mockito.lenient().when(userRepository.isUsernameAlreadyExists(any(String.class))).thenRetur
n(false);
}
```
Пожалуйста, обратите внимание на использование *Mockito.lenient ()* здесь. *Mockito* создает исключение *UnsupportedStubbingException,* когда инициализированный макет не вызывается одним из методов теста во время выполнения. Мы можем избежать этой строгой проверки заглушки, используя этот метод при инициализации макетов.

Мы можем даже вставить фиктивный объект в параметр метода тестирования:

```
@Test
void givenValidUser__whenSaveUser__thenSucceed(@Mock MailClient mailClient) {
   //Given
    user = new User("Jerry", 12);
    when(userRepository.insert(any(User.class))).then(new Answer<User>() {
        int sequence = 1;
```

```
        @Override
        public User answer(InvocationOnMock invocation) throws Throwable {
            User user = (User) invocation.getArgument(0);
            user.setId(sequence++);
            return user;
        }
    });

    userService = new DefaultUserService(userRepository, settingRepository, mailClient);

    //When
    User insertedUser = userService.register(user);

    //Then
    verify(userRepository).insert(user);
    Assertions.assertNotNull(user.getId());
    verify(mailClient).sendUserRegistrationMail(insertedUser);
}
```

Обратите внимание, что макет *MailClient* , который мы вводим в качестве тестового параметра, НЕ будет тем же экземпляром, который мы добавили в метод *init* .

5. Заключение

Junit 5 предоставил хорошую модель для расширения. Мы продемонстрировали простое расширение Mockito, которое упростило нашу логику создания макетов

Conditional Test Execution

The `ExecutionCondition` extension API in JUnit Jupiter allows developers to either *enable* or *disable* a container or test based on certain conditions *programmatically*. The simplest example of such a condition is the built-in `DisabledCondition` which supports the `@Disabled` annotation (see Disabling Tests). In addition to `@Disabled`, JUnit Jupiter also supports several other annotation-based conditions in the `org.junit.jupiter.api.condition` package that allow developers to enable or disable containers and tests *declaratively*. When multiple `ExecutionCondition` extensions are registered, a container or test is disabled as soon as one of the conditions returns *disabled*. If you wish to provide details about why they might be disabled, every annotation associated with these built-in conditions has a `disabledReason` attribute available for that purpose.

See `ExecutionCondition` and the following sections for details.

> *Composed Annotations*
>
> Note that any of the *conditional* annotations listed in the following sections may also be used as a meta-annotation in order to create a custom *composed annotation*. For example, the `@TestOnMac` annotation in the @EnabledOnOs demo shows how you can combine `@Test` and `@EnabledOnOs` in a single, reusable annotation.
>
> Unless otherwise stated, each of the *conditional* annotations listed in the following sections can only be declared once on a given test interface, test class, or test method. If a conditional annotation is directly present, indirectly present, or meta-present multiple times on a given element, only the first such annotation discovered by JUnit will be used; any additional declarations will be silently ignored. Note, however, that each conditional annotation may be used in conjunction with other conditional annotations in the `org.junit.jupiter.api.condition` package.

2.7.1. Operating System Conditions

A container or test may be enabled or disabled on a particular operating system via the `@EnabledOnOs` and `@DisabledOnOs` annotations.

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

@TestOnMac
void testOnMac() {
    // ...
}

@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Test
@EnabledOnOs(MAC)
@interface TestOnMac {
}
```

2.7.2. Java Runtime Environment Conditions

A container or test may be enabled or disabled on particular versions of the Java Runtime Environment (JRE) via the `@EnabledOnJre` and `@DisabledOnJre` annotations or on a particular range of versions of the JRE via the `@EnabledForJreRange` and `@DisabledForJreRange` annotations. The range defaults to `JRE.JAVA_8` as the lower border (`min`) and `JRE.OTHER` as the higher border (`max`), which allows usage of half open ranges.

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}

@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_9)
void fromJava9toCurrentJavaFeatureNumber() {
    // ...
}

@Test
@EnabledForJreRange(max = JAVA_11)
void fromJava8To11() {
    // ...
}

@Test
```

```
@DisabledOnJre(JAVA_9)
void notOnJava9() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_9, max = JAVA_11)
void notFromJava9to11() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_9)
void notFromJava9toCurrentJavaFeatureNumber() {
    // ...
}

@Test
@DisabledForJreRange(max = JAVA_11)
void notFromJava8to11() {
    // ...
}
```

2.7.3. System Property Conditions

A container or test may be enabled or disabled based on the value of the `named` JVM system property via the `@EnabledIfSystemProperty` and `@DisabledIfSystemProperty` annotations. The value supplied via the `matches` attribute will be interpreted as a regular expression.

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```

> As of JUnit Jupiter 5.6, `@EnabledIfSystemProperty` and `@DisabledIfSystemProperty` are *repeatable annotations*. Consequently, these annotations may be declared multiple times on a test interface, test class, or test method. Specifically, these annotations will be found if they are directly present, indirectly present, or meta-present on a given element.

2.7.4. Environment Variable Conditions

A container or test may be enabled or disabled based on the value of the `named` environment variable from the underlying operating system via the `@EnabledIfEnvironmentVariable` and `@DisabledIfEnvironmentVariable` annotations. The value supplied via the `matches` attribute will be interpreted as a regular expression.

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
    // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
    // ...
}
```

> As of JUnit Jupiter 5.6, `@EnabledIfEnvironmentVariable` and `@DisabledIfEnvironmentVariable` are *repeatable annotations*. Consequently, these annotations may be declared multiple times on a test interface, test class, or test method. Specifically, these annotations will be found if they are directly present, indirectly present, or meta-present on a given element.

2.7.5. Custom Conditions

A container or test may be enabled or disabled based on the boolean return of a method via the `@EnabledIf` and `@DisabledIf` annotations. The method is provided to the annotation via its name, or its fully qualified name if located outside the test class. If needed, the condition method can take a single parameter of type `ExtensionContext`.

```
@Test
@EnabledIf("customCondition")
void enabled() {
    // ...
}

@Test
@DisabledIf("customCondition")
void disabled() {
    // ...
}

boolean customCondition() {
    return true;
}
```

Parameterized Tests

Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular `@Test` methods but use the `@ParameterizedTest` annotation instead. In addition, you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.

The following example demonstrates a parameterized test that uses the `@ValueSource` annotation to specify a `String` array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

When executing the above parameterized test method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
palindromes(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

> Parameterized tests are currently an *experimental* feature. Consult the table in Experimental APIs for details.

2.15.1. Required Setup

In order to use parameterized tests you need to add a dependency on the `junit-jupiter-params` artifact. Please refer to Dependency Metadata for details.

2.15.2. Consuming Arguments

Parameterized test methods typically *consume* arguments directly from the configured source (see Sources of Arguments) following a one-to-one correlation between argument source index and method parameter index (see examples in @CsvSource). However, a parameterized test method may also choose to *aggregate* arguments from the source into a single object passed to the method (see Argument Aggregation). Additional arguments may also be provided by a `ParameterResolver` (e.g., to obtain an instance of `TestInfo`, `TestReporter`, etc.). Specifically, a parameterized test method must declare formal parameters according to the following rules.

- Zero or more *indexed arguments* must be declared first.
- Zero or more *aggregators* must be declared next.
- Zero or more arguments supplied by a `ParameterResolver` must be declared last.

In this context, an *indexed argument* is an argument for a given index in the `Arguments` provided by an `ArgumentsProvider` that is passed as an argument to the parameterized method at the same index in the method's formal parameter list. An *aggregator* is any parameter of type `ArgumentsAccessor` or any parameter annotated with `@AggregateWith`.

2.15.3. Sources of Arguments

Out of the box, JUnit Jupiter provides quite a few *source* annotations. Each of the following subsections provides a brief overview and an example for each of them. Please refer to the Javadoc in the `org.junit.jupiter.params.provider` package for additional information.

@ValueSource

`@ValueSource` is one of the simplest possible sources. It lets you specify a single array of literal values and can only be used for providing a single argument per parameterized test invocation.

The following types of literal values are supported by `@ValueSource`.

- `short`
- `byte`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`
- `java.lang.String`
- `java.lang.Class`

For example, the following `@ParameterizedTest` method will be invoked three times, with the values `1`, `2`, and `3` respectively.

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

Null and Empty Sources

In order to check corner cases and verify proper behavior of our software when it is supplied *bad input*, it can be useful to have `null` and *empty* values supplied to our parameterized tests. The following annotations serve as sources of `null` and empty values for parameterized tests that accept a single argument.

- `@NullSource`: provides a single `null` argument to the annotated `@ParameterizedTest` method.
  - `@NullSource` cannot be used for a parameter that has a primitive type.
- `@EmptySource`: provides a single *empty* argument to the annotated `@ParameterizedTest` method for parameters of the following types: `java.lang.String`, `java.util.List`, `java.util.Set`, `java.util.Map`, primitive arrays (e.g., `int[]`, `char[][]`, etc.), object arrays (e.g.,`String[]`, `Integer[][]`, etc.).

- Subtypes of the supported types are not supported.

- `@NullAndEmptySource`: a *composed annotation* that combines the functionality of `@NullSource` and `@EmptySource`.

If you need to supply multiple varying types of *blank* strings to a parameterized test, you can achieve that using `@ValueSource` — for example, `@ValueSource(strings = {" ", "   ", "\t", "\n"})`.

You can also combine `@NullSource`, `@EmptySource`, and `@ValueSource` to test a wider range of `null`, *empty*, and *blank* input. The following example demonstrates how to achieve this for strings.

```
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
```

Making use of the composed `@NullAndEmptySource` annotation simplifies the above as follows.

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
```

> Both variants of the `nullEmptyAndBlankStrings(String)` parameterized test method result in six invocations: 1 for `null`, 1 for the empty string, and 4 for the explicit blank strings supplied via `@ValueSource`.

@EnumSource

`@EnumSource` provides a convenient way to use `Enum` constants.

```
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void testWithEnumSource(TemporalUnit unit) {
    assertNotNull(unit);
}
```

The annotation's `value` attribute is optional. When omitted, the declared type of the first method parameter is used. The test will fail if it does not reference an enum type. Thus, the `value` attribute is required in the above example because the method parameter is declared as `TemporalUnit`, i.e. the interface implemented by `ChronoUnit`, which isn't an enum type. Changing the method parameter type to `ChronoUnit` allows you to omit the explicit enum type from the annotation as follows.

```
@ParameterizedTest
@EnumSource
void testWithEnumSourceWithAutoDetection(ChronoUnit unit) {
    assertNotNull(unit);
}
```

The annotation provides an optional `names` attribute that lets you specify which constants shall be used, like in the following example. If omitted, all constants will be used.

```
@ParameterizedTest
@EnumSource(names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(ChronoUnit unit) {
    assertTrue(EnumSet.of(ChronoUnit.DAYS, ChronoUnit.HOURS).contains(unit));
}
```

The `@EnumSource` annotation also provides an optional `mode` attribute that enables fine-grained control over which constants are passed to the test method. For example, you can exclude names from the enum constant pool or specify regular expressions as in the following examples.

```
@ParameterizedTest
```

```
@EnumSource(mode = EXCLUDE, names = { "ERAS", "FOREVER" })
void testWithEnumSourceExclude(ChronoUnit unit) {
    assertFalse(EnumSet.of(ChronoUnit.ERAS, ChronoUnit.FOREVER).contains(unit));
}
@ParameterizedTest
@EnumSource(mode = MATCH_ALL, names = "^.*DAYS$")
void testWithEnumSourceRegex(ChronoUnit unit) {
    assertTrue(unit.name().endsWith("DAYS"));
}
```

@MethodSource

`@MethodSource` allows you to refer to one or more *factory* methods of the test class or external classes.

Factory methods within the test class must be `static` unless the test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`; whereas, factory methods in external classes must always be `static`. In addition, such factory methods must not accept any arguments.

Each factory method must generate a *stream* of *arguments*, and each set of arguments within the stream will be provided as the physical arguments for individual invocations of the annotated `@ParameterizedTest` method. Generally speaking this translates to a `Stream` of `Arguments` (i.e., `Stream<Arguments>`); however, the actual concrete return type can take on many forms. In this context, a "stream" is anything that JUnit can reliably convert into a `Stream`, such as `Stream`, `DoubleStream`, `LongStream`, `IntStream`, `Collection`, `Iterator`, `Iterable`, an array of objects, or an array of primitives. The "arguments" within the stream can be supplied as an instance of `Arguments`, an array of objects (e.g., `Object[]`), or a single value if the parameterized test method accepts a single argument.

If you only need a single parameter, you can return a `Stream` of instances of the parameter type as demonstrated in the following example.

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

If you do not explicitly provide a factory method name via `@MethodSource`, JUnit Jupiter will search for a *factory* method that has the same name as the current `@ParameterizedTest` method by convention. This is demonstrated in the following example.

```
@ParameterizedTest
@MethodSource
void testWithDefaultLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> testWithDefaultLocalMethodSource() {
    return Stream.of("apple", "banana");
}
```

Streams for primitive types (`DoubleStream`, `IntStream`, and `LongStream`) are also supported as demonstrated by the following example.

```
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertNotEquals(9, argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip(10);
}
```

If a parameterized test method declares multiple parameters, you need to return a collection, stream, or array of `Arguments` instances or object arrays as shown below (see the Javadoc for `@MethodSource` for further details on supported return types). Note that `arguments(Object…)` is a static factory method defined in the `Arguments` interface. In addition, `Arguments.of(Object…)` may be used as an alternative to `arguments(Object…)`.

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}
```

An external, `static` *factory* method can be referenced by providing its *fully qualified method name* as demonstrated in the following example.

```
package example;

import java.util.stream.Stream;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class ExternalMethodSourceDemo {

    @ParameterizedTest
    @MethodSource("example.StringsProviders#tinyStrings")
    void testWithExternalMethodSource(String tinyString) {
        // test with tiny string
    }
}

class StringsProviders {

    static Stream<String> tinyStrings() {
        return Stream.of(".", "oo", "OOO");
    }

}
```

Передовой опыт тестирования в Java

Автор оригинала: Philipp Hauer

- Блог компании FunCorp,
- Java,
- Тестирование веб-сервисов,
- Kotlin
- Перевод

Чтобы покрытие кода было достаточным, а создание нового функционала и рефакторинг старого проходили без страха что-то сломать, тесты должны быть поддерживаемыми и легко читаемыми. В этой статье я расскажу о множестве приёмов написания юнит- и интеграционных тестов на Java, собранных мной за несколько лет. Я буду опираться на современные технологии: JUnit5, AssertJ, Testcontainers, а также не обойду вниманием Kotlin. Некоторые советы покажутся вам очевидными, другие могут идти вразрез с тем, что вы читали в книгах о разработке ПО и тестировании.

Вкратце

- Пишите тесты кратко и конкретно, используя вспомогательные функции, параметризацию, разнообразные примитивы библиотеки AssertJ, не злоупотребляйте переменными, проверяйте только то, что относится к тестируемому функционалу и не засовывайте все нестандартные случаи в один тест
- Пишите самодостаточные тесты, раскрывайте все релевантные параметры, вставляйте тестовые данные прямо внутрь тестов и вместо наследования пользуйтесь композицией
- Пишите прямолинейные тесты, чтобы не переиспользовать продакшн-код, сравнивайте выдачу тестируемых методов с константами прямо в коде теста
- KISS важнее DRY
- Запускайте тесты в среде, максимально похожей на боевую, тестируйте максимально полную связку компонентов, не используйте in-memory-базы данных
- JUnit5 и AssertJ — очень хороший выбор
- Вкладывайтесь в простоту тестирования: избегайте статических свойств и методов, используйте внедрение в конструкторы, используйте экземпляры класса Clock и отделяйте бизнес-логику от асинхронной.

Общие положения

Given, When, Then (Дано, Когда, То)

Тест должен содержать три блока, разделённых пустыми строками. Каждый блок должен быть максимально коротким. Используйте локальные методы для компактности записи.

Given / Дано (ввод): подготовка теста, например, создание данных и конфигурация моков.
When / Когда (действие): вызов тестируемого метода
Then / То (вывод): проверка корректности полученного значения

```
// Правильно
@Test
public void findProduct() {
    insertIntoDatabase(new Product(100, "Smartphone"));
    Product product = dao.findProduct(100);
    assertThat(product.getName()).isEqualTo("Smartphone");
}
```

Используйте префиксы “actual*” и “expected*”

```
// Неправильно
ProductDTO product1 = requestProduct(1);
ProductDTO product2 = new ProductDTO("1", List.of(State.ACTIVE, State.REJECTED))
assertThat(product1).isEqualTo(product2);
```

Если вы собираетесь использовать переменные в проверке на совпадение значений, добавьте к этим переменным префиксы “actual” и “expected”. Так вы улучшите читаемость кода и проясните назначение переменных. Кроме того, так их сложнее перепутать при сравнении.

```
// Правильно
ProductDTO actualProduct = requestProduct(1);
ProductDTO expectedProduct = new ProductDTO("1", List.of(State.ACTIVE, State.REJEC
TED))
```

```
assertThat(actualProduct).isEqualTo(expectedProduct); // ясно и красиво
```

Используйте заданные значения вместо случайных

Избегайте подавать случайные значения на вход тестов. Это может привести к «морганию» тестов, что чертовски сложно отлаживать. Кроме того, увидев в сообщении об ошибке случайное значение, вы не сможете проследить его до того места, где ошибка возникла.

```
// Неправильно
Instant ts1 = Instant.now(); // 1557582788
Instant ts2 = ts1.plusSeconds(1); // 1557582789
int randomAmount = new Random().nextInt(500); // 232
UUID uuid = UUID.randomUUID(); // d5d1f61b-0a8b-42be-b05a-bd458bb563ad
```

Используйте для всего подряд разные заранее заданные значения. Так вы получите идеально воспроизводимые результаты тестов, а также быстро найдёте нужное место в коде по сообщению об ошибке.

```
// Правильно
Instant ts1 = Instant.ofEpochSecond(1550000001);
Instant ts2 = Instant.ofEpochSecond(1550000002);
int amount = 50;
UUID uuid = UUID.fromString("00000000-000-0000-0000-000000000001");
```

Вы можете записать это ещё короче, используя вспомогательные функции (см. ниже).

Пишите краткие и конкретные тесты

Где можно, используйте вспомогательные функции

Вычленяйте повторяющийся код в локальные функции и давайте им понятные имена. Так ваши тесты будут компактными и легко читаемыми с первого взгляда.

```
// Неправильно
@Test
public void categoryQueryParameter() throws Exception {
    List<ProductEntity> products = List.of(
            new ProductEntity().setId("1").setName("Envelope").setCategory("Office
").setDescription("An Envelope").setStockAmount(1),
            new ProductEntity().setId("2").setName("Pen").setCategory("Office").se
tDescription("A Pen").setStockAmount(1),
            new ProductEntity().setId("3").setName("Notebook").setCategory("Hardwa
re").setDescription("A Notebook").setStockAmount(2)
    );
    for (ProductEntity product : products) {
        template.execute(createSqlInsertStatement(product));
    }

    String responseJson = client.perform(get("/products?category=Office"))
            .andExpect(status().is(200))
            .andReturn().getResponse().getContentAsString();
    assertThat(toDTOs(responseJson))
            .extracting(ProductDTO::getId)
            .containsOnly("1", "2");
}
```

```
// Правильно
@Test
public void categoryQueryParameter2() throws Exception {
    insertIntoDatabase(
            createProductWithCategory("1", "Office"),
            createProductWithCategory("2", "Office"),
            createProductWithCategory("3", "Hardware")
    );

    String responseJson = requestProductsByCategory("Office");

    assertThat(toDTOs(responseJson))
            .extracting(ProductDTO::getId)
            .containsOnly("1", "2");
}
```

- используйте вспомогательные функции для создания данных (объектов) (`createProductWithCategory()`) и сложных проверок. Передавайте во вспомогательные функции только те параметры, которые релевантны в этом тесте, для остальных используйте адекватные значения по умолчанию. В Kotlin для этого есть дефолтные значения параметров, а в Java можно использовать цепочки вызова методов и перегрузку для имитации дефолтных параметров
- список параметров переменной длины сделает ваш код ещё изящнее (`insertIntoDatabase()`)
- вспомогательные функции также можно использовать для создания простых значений. В Kotlin это сделано ещё лучше через функции-расширения

```
// Правильно (Java)
Instant ts = toInstant(1); // Instant.ofEpochSecond(1550000001)
UUID id = toUUID(1); // UUID.fromString("00000000-0000-0000-a000-000000000001")
```

```
// Правильно (Kotlin)
val ts = 1.toInstant()
val id = 1.toUUID()
```

Вспомогательные функции на Kotlin можно реализовать так:

```
fun Int.toInstant(): Instant = Instant.ofEpochSecond(this.toLong())
```

```
fun Int.toUUID(): UUID = UUID.fromString("00000000-0000-0000-a000-${this.toString(
).padStart(11, '0')}")
```

Не злоупотребляйте переменными

Условный рефлекс у программиста — вынести часто используемые значения в переменные.

```
// Неправильно
@Test
public void variables() throws Exception {
    String relevantCategory = "Office";
    String id1 = "4243";
    String id2 = "1123";
    String id3 = "9213";
    String irrelevantCategory = "Hardware";
    insertIntoDatabase(
            createProductWithCategory(id1, relevantCategory),
            createProductWithCategory(id2, relevantCategory),
            createProductWithCategory(id3, irrelevantCategory)
```

```
        );
        String responseJson = requestProductsByCategory(relevantCategory);
        assertThat(toDTOs(responseJson))
                .extracting(ProductDTO::getId)
                .containsOnly(id1, id2);
    }
```

Увы, это очень перегружает код. Хуже того, увидев значение в сообщении об ошибке, — его будет невозможно проследить до места, где ошибка возникла.

«KISS важнее DRY»

```
    // Правильно
    @Test
    public void variables() throws Exception {
        insertIntoDatabase(
                createProductWithCategory("4243", "Office"),
                createProductWithCategory("1123", "Office"),
                createProductWithCategory("9213", "Hardware")
        );
        String responseJson = requestProductsByCategory("Office");
        assertThat(toDTOs(responseJson))
                .extracting(ProductDTO::getId)
                .containsOnly("4243", "1123");
    }
```

Если вы стараетесь писать тесты максимально компактно (что я, в любом случае, горячо рекомендую), то переиспользуемые значения хорошо видны. Сам код становится более

убористым и хорошо читаемым. И, наконец, сообщение об ошибке приведёт вас точно к той строке, где ошибка возникла.

Не расширяйте существующие тесты, чтобы «добавить ещё одну маленькую штучку»

```
// Неправильно
public class ProductControllerTest {
    @Test
    public void happyPath() {
        // здесь много кода...
    }
}
```

Всегда есть соблазн добавить частный случай к существующему тесту, проверяющему базовую функциональность. Но в результате тесты становятся больше и сложнее для понимания. Частные случаи, раскиданные по большой простыне кода, легко не заметить. Если тест сломался, вы не сразу поймёте, что именно послужило причиной.

```
// Правильно
public class ProductControllerTest {
    @Test
    public void multipleProductsAreReturned() {}
    @Test
    public void allProductValuesAreReturned() {}
    @Test
    public void filterByCategory() {}
```

```
    @Test
    public void filterByDateCreated() {}
}
```

Вместо этого напишите новый тест с наглядным названием, из которого сразу будет понятно, какого поведения он ожидает от тестируемого кода. Да, придётся набрать больше букв на клавиатуре (против этого, напомню, хорошо способствуют вспомогательные функции), но зато вы получите простой и понятный тест с предсказуемым результатом. Это, кстати, отличный способ документировать новый функционал.

Проверяйте только то, что хотите протестировать

Думайте о том функционале, который тестируется. Избегайте делать лишние проверки просто потому, что есть такая возможность. Более того, помните о том, что уже проверялось в ранее написанных тестах и не проверяйте это повторно. Тесты должны быть компактными и их ожидаемое поведение должно быть очевидным и лишённым ненужных подробностей.

Предположим, что мы хотим проверить HTTP-ручку, возвращающую список товаров. Наш тестовый набор должен содержать следующие тесты:

1. Один большой тест маппинга, который проверяет, что все значения из БД корректно возвращаются в JSON-ответе и правильно присваиваются в нужном формате. Мы легко можем написать это при помощи функций `isEqualTo()` (для единичного элемента) или `containsOnly()` (для множества элементов) из пакета AssertJ, если вы правильно реализуете метод `equals()`.

```
String responseJson = requestProducts();
```

```
ProductDTO expectedDTO1 = new ProductDTO("1", "envelope", new Category("office"),
```

```
ProductDTO expectedDTO2 = new ProductDTO("2", "envelope", new Category("smartphone
"), List.of(States.ACTIVE));
```

```
assertThat(toDTOs(responseJson))
        .containsOnly(expectedDTO1, expectedDTO2);
```

2. Несколько тестов, проверяющих корректное поведение параметра ?category. Здесь мы хотим

проверить только правильную работу фильтров, а не значения свойств, потому что мы сделали это раньше. Следовательно, нам достаточно проверить совпадения полученных id товаров:

```
String responseJson = requestProductsByCategory("Office");
```

```
assertThat(toDTOs(responseJson))
        .extracting(ProductDTO::getId)
        .containsOnly("1", "2");
```

3. Ещё пару тестов, проверяющих особые случаи или особую бизнес-логику, например что определённые значения в ответе вычислены корректно. В этом случае нас интересуют только несколько полей из всего JSON-ответа. Тем самым мы своим тестом документируем именно эту специальную логику. Понятно, что ничего кроме этих полей нам здесь не нужно.

```
assertThat(actualProduct.getPrice()).isEqualTo(100);
```

Самодостаточные тесты

Не прячьте релевантные параметры (во вспомогательных функциях)

```
// Неправильно
insertIntoDatabase(createProduct());
List<ProductDTO> actualProducts = requestProductsByCategory();
assertThat(actualProducts).containsOnly(new ProductDTO("1", "Office"));
```

Использовать вспомогательные функции для генерации данных и проверки условий удобно, но их следует вызывать с параметрами. Принимайте на вход параметры для всего, что значимо в рамках теста и должно контролироваться из кода теста. Не заставляйте читателя переходить внутрь вспомогательной функции, чтобы понять смысл теста. Простое правило: смысл теста должен быть понятен при взгляде на сам тест.

```
// Правильно
```

```
insertIntoDatabase(createProduct("1", "Office"));
```

```
List<ProductDTO> actualProducts = requestProductsByCategory("Office");
```

```
assertThat(actualProducts).containsOnly(new ProductDTO("1", "Office"));
```

Держите тестовые данные внутри самих тестов

Всё должно быть внутри. Велик соблазн перенести часть данных в метод `@Before` и переиспользовать их оттуда. Но это вынудит читателя скакать туда-сюда по файлу, чтобы понять, что именно тут происходит. Опять же, вспомогательные функции помогут избежать повторений и сделают тесты более понятными.

Используйте композицию вместо наследования

Не выстраивайте сложных иерархий тестовых классов.

```
// Неправильно
```

```
class SimpleBaseTest {}
```

```
class AdvancedBaseTest extends SimpleBaseTest {}
```

```
class AllInklusiveBaseTest extends AdvancedBaseTest {}
```

```
class MyTest extends AllInklusiveBaseTest {}
```

Такие иерархии усложняют понимание и вы, скорее всего, быстро обнаружите себя пишущим очередного наследника базового теста, внутри которого зашито множество хлама, который текущему тесту вовсе не нужен. Это отвлекает читателя и приводит к трудноуловимым ошибкам. Наследование не гибко: как вы сами думаете, можно ли использовать все методы класса `AllInclusiveBaseTest`, но ни одного из его родительского `AdvancedBaseTest`? Более того, читателю придётся постоянно прыгать между различными базовыми классами, чтобы понять общую картину.

«Лучше продублировать код, чем выбрать неправильную абстракцию» (Sandi Metz)

Вместо этого я рекомендую использовать композицию. Напишите маленькие фрагменты кода и классы для каждой задачи, связанной с фикстурами (запустить тестовую базу данных, создать схему, вставить данные, запустить мок-сервер). Переиспользуйте эти запчасти в методе `@BeforeAll` или через присвоение созданных объектов полям тестового класса. Таким образом, вы сможете собирать каждый новый тестовый класс из этих заготовок, как из деталей

Лего. В результате каждый тест будет иметь свой собственный понятный набор фикстур и гарантировать, что в нём не происходит ничего постороннего. Тест становится самодостаточным, потому что содержит в себе всё необходимое.

```
// Правильно
public class MyTest {
    // композиция вместо наследования
    private JdbcTemplate template;
    private MockWebServer taxService;

    @BeforeAll
    public void setupDatabaseSchemaAndMockWebServer() throws IOException {
        this.template = new DatabaseFixture().startDatabaseAndCreateSchema();
        this.taxService = new MockWebServer();
        taxService.start();
    }
}

// В другом файле
public class DatabaseFixture {
    public JdbcTemplate startDatabaseAndCreateSchema() throws IOException {
        PostgreSQLContainer db = new PostgreSQLContainer("postgres:11.2-alpine");
        db.start();
        DataSource dataSource = DataSourceBuilder.create()
                .driverClassName("org.postgresql.Driver")
                .username(db.getUsername())
                .password(db.getPassword())
                .url(db.getJdbcUrl())
```

```
                .build();
        JdbcTemplate template = new JdbcTemplate(dataSource);
        SchemaCreator.createSchema(template);
        return template;
    }
}
```

И ещё раз:

«KISS важнее DRY»

Прямолинейные тесты — это хорошо. Сравнивайте результат с константами

Не переиспользуйте продакшн-код

Тесты должны проверять продакшн-код, а не переиспользовать его. Если вы переиспользуете боевой код в тесте, вы можете пропустить баг в этом коде, потому что больше не тестируете его.

```
// Неправильно
boolean isActive = true;
boolean isRejected = true;
insertIntoDatabase(new Product(1, isActive, isRejected));

ProductDTO actualDTO = requestProduct(1);

// переиспользование боевого кода
List<State> expectedStates = ProductionCode.mapBooleansToEnumList(isActive, isReje
cted);
```

```
assertThat(actualDTO.states).isEqualTo(expectedStates);
```

Вместо этого при написании тестов думайте в терминах ввода и вывода. Тест подаёт данные на вход и сравнивает вывод с предопределёнными константами. Бóльшую часть времени переиспользование кода не требуется.

```
// Do
assertThat(actualDTO.states).isEqualTo(List.of(States.ACTIVE, States.REJECTED));
```

Не копируйте бизнес-логику в тесты

Маппинг объектов — яркий пример случая, когда тесты тащат в себя логику из боевого кода. Предположим наш тест содержит метод `mapEntityToDto()`, результат выполнения которого используется для проверки, что полученный DTO содержит те же значения, что и элементы, которые были добавлены в базу в начале теста. В этом случае вы, скорее всего, скопируете в тест боевой код, который может содержать ошибки.

```
// Неправильно
ProductEntity inputEntity = new ProductEntity(1, "envelope", "office", false, true
, 200, 10.0);
insertIntoDatabase(input);

ProductDTO actualDTO = requestProduct(1);

// mapEntityToDto() содержит ту же логику, что и продакшн-код
ProductDTO expectedDTO = mapEntityToDto(inputEntity);
assertThat(actualDTO).isEqualTo(expectedDTO);
```

Правильным будет решение, при котором `actualDTO` сравнивается с созданным вручную эталонным объектом с заданными значениями. Это предельно просто, понятно и защищает от потенциальных ошибок.

```
// Правильно
```

```
ProductDTO expectedDTO = new ProductDTO("1", "envelope", new Category("office"), L
```

```
assertThat(actualDTO).isEqualTo(expectedDTO);
```

Если вы не хотите создавать и проверять на совпадение целый эталонный объект, можете проверить дочерний объект или вообще только релевантные тесту свойства объекта.

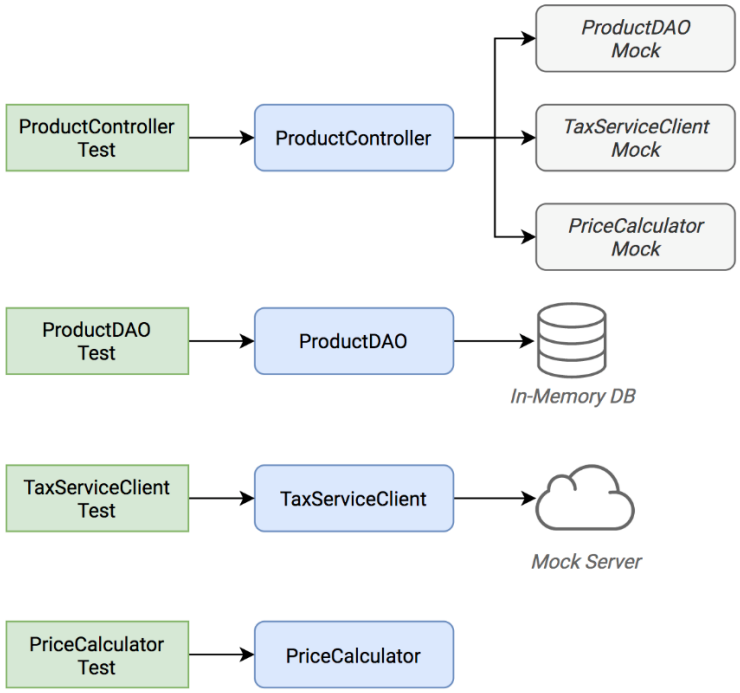
Не пишите слишком много логики

Напомню, что тестирование касается в основном ввода и вывода. Подавайте на вход данные и проверяйте, что вам вернулось. Нет необходимости писать сложную логику внутри тестов. Если вы вводите в тест циклы и условия, вы делаете его менее понятным и более неустойчивым к ошибкам. Если ваша логика проверки сложна, пользуйтесь многочисленными функциями AssertJ, которые сделают эту работу за вас.

Запускайте тесты в среде, максимально похожей на боевую

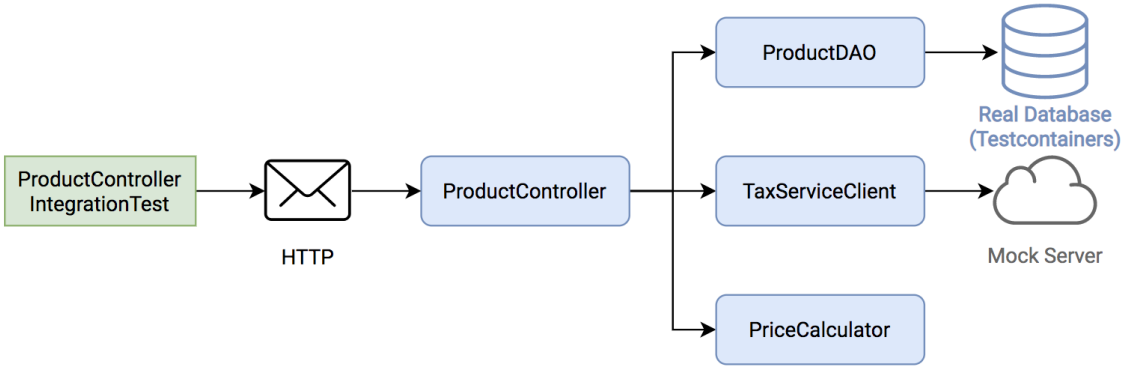
Тестируйте максимально полную связку компонентов

Обычно рекомендуется тестировать каждый класс изолированно при помощи моков. У этого подхода, однако, есть и недостатки: таким образом не тестируется взаимодействие классов между собой, и любой рефакторинг общих сущностей сломает все тесты разом, потому что у каждого внутреннего класса свои тесты. Кроме того, если писать тесты для каждого класса, то их будет просто слишком много.



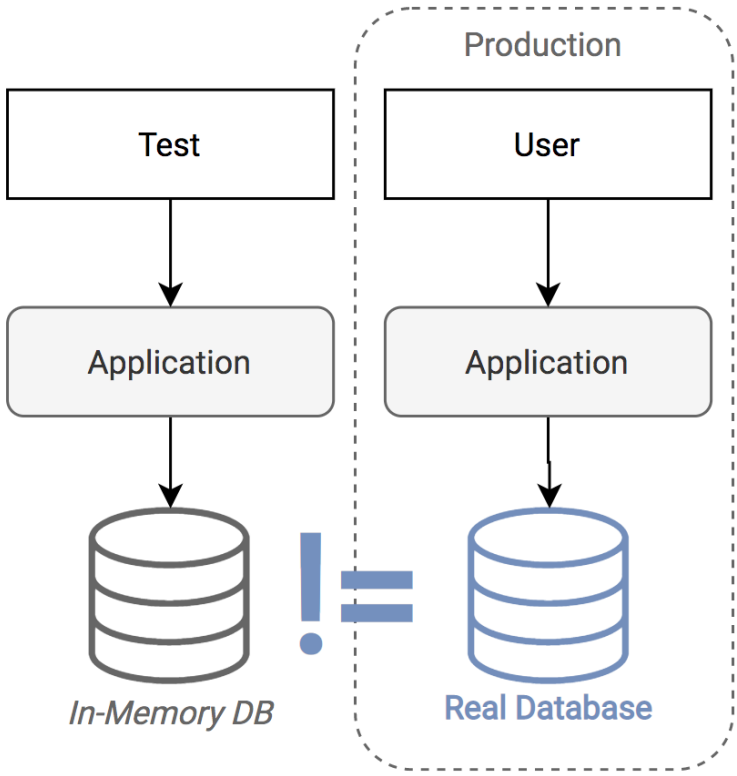
Изолированное юнит-тестирование каждого класса

Вместо этого я рекомендую сосредоточиться на интеграционном тестировании. Под «интеграционным тестированием» я подразумеваю сбор всех классов воедино (как на продакшене) и тестирование всей связки, включая инфраструктурные компоненты (HTTP-сервер, базу данных, бизнес-логику). В этом случае вы тестируете поведение вместо реализации. Такие тесты более аккуратны, близки к реальному миру и устойчивы к рефакторингу внутренних компонентов. В идеале, вам будет достаточно одного класса тестов.



Интеграционное тестирование (= собрать все классы вместе и тестировать связку)

Не используйте in-memory-базы данных для тестов



С in-memory базой вы тестируете не в той среде, где будет работать ваш код

Используя in-memory базу (H2, HSQLDB, Fongo) для тестов, вы жертвуете их достоверностью и рамками применимости. Такие базы данных часто ведут себя иначе и выдают отличающиеся результаты. Такой тест может пройти успешно, но не гарантирует корректной работы приложения на проде. Более того, вы можете запросто оказаться в ситуации, когда вы не можете использовать или протестировать какое-то характерное для вашей базы поведение или фичу, потому в in-memory БД они не реализованы или ведут себя иначе.

Решение: использовать такую же БД, как и в реальной эксплуатации. Замечательная библиотека Testcontainers предоставляет богатый API для Java-приложений, позволяющий управлять контейнерами прямо из кода тестов.

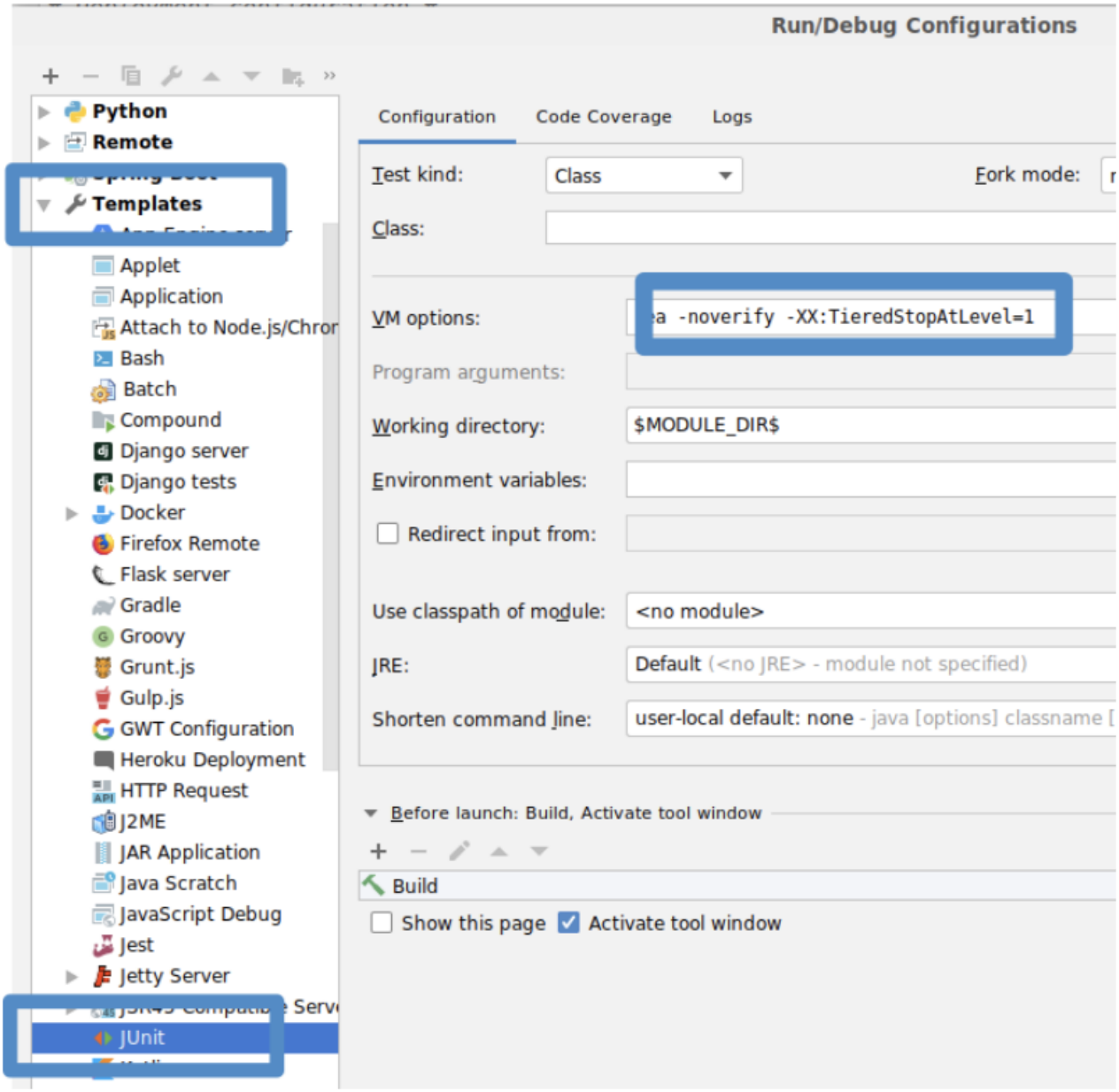
Java/JVM

Используйте `-noverify -XX:TieredStopAtLevel=1`

Всегда добавляйте опции `JVM -noverify -XX:TieredStopAtLevel=1` в вашу конфигурацию для запуска тестов. Это сэкономит вам 1-2 секунды на старте виртуальной машины перед тем, как начнётся выполнение тестов. Это особенно полезно на начальной стадии работы над тестами, когда вы часто запускаете их из IDE.

Обратите внимание, что начиная с Java 13 `-noverify` объявлен устаревшим.

Совет: добавьте эти аргументы к шаблону конфигурации “JUnit” в IntelliJ IDEA, чтобы не делать это каждый раз при создании нового проекта.



Используйте AssertJ

AssertJ — исключительно мощная и зрелая библиотека, обладающая развитым и безопасным API, а также большим набором функций проверки значений и информативных сообщений об ошибках тестирования. Множество удобных функций проверки избавляет программиста от необходимости описывать комплексную логику в теле тестов, позволяя делать тесты лаконичными. Например:

```
assertThat(actualProduct)
```

```
        .isEqualToIgnoringGivenFields(expectedProduct, "id");
```

```
assertThat(actualProductList).containsExactly(
```

```
        createProductDTO("1", "Smartphone", 250.00),
        createProductDTO("1", "Smartphone", 250.00)
);

assertThat(actualProductList)
        .usingElementComparatorIgnoringFields("id")
        .containsExactly(expectedProduct1, expectedProduct2);

assertThat(actualProductList)
        .extracting(Product::getId)
        .containsExactly("1", "2");

assertThat(actualProductList)
        .anySatisfy(product -> assertThat(product.getDateCreated()).isBetween(inst
ant1, instant2));

assertThat(actualProductList)
        .filteredOn(product -> product.getCategory().equals("Smartphone"))
        .allSatisfy(product -> assertThat(product.isLiked()).isTrue());
```

Избегайте использовать assertTrue() и assertFalse()

Использование простых `assertTrue()` или `assertFalse()` приводит к загадочным сообщениям об ошибках тестов:

```
// Неправильно
assertTrue(actualProductList.contains(expectedProduct));
```

```
assertTrue(actualProductList.size() == 5);
```

```
assertTrue(actualProduct instanceof Product);
```

```
expected: <true> but was: <false>
```

Используйте вместо них вызовы AssertJ, которые «из коробки» возвращают понятные и информативные сообщения.

```
// Правильно
```

```
assertThat(actualProductList).contains(expectedProduct);
```

```
assertThat(actualProductList).hasSize(5);
```

```
assertThat(actualProduct).isInstanceOf(Product.class);
```

```
Expecting:
```

```
 <[Product[id=1, name='Samsung Galaxy']]>
```

```
to contain:
```

```
 <[Product[id=2, name='iPhone']]>
```

```
but could not find:
```

```
 <[Product[id=2, name='iPhone']]>
```

Если вам надо проверить boolean-значение, сделайте сообщение более информативным при помощи метода `as()` AssertJ.

Используйте JUnit5

JUnit5 — превосходная библиотека для (юнит-)тестирования. Она находится в процессе постоянного развития и предоставляет программисту множество полезных возможностей, таких, например, как параметризованные тесты, группировки, условные тесты, контроль жизненного цикла.

Используйте параметризованные тесты

Параметризованные тесты позволяют запускать один и тот же тест с набором различных входных значений. Это позволяет проверять несколько кейсов без написания лишнего кода. В JUnit5 для этого есть отличные инструменты `@ValueSource`, `@EnumSource`, `@CsvSource` и `@MethodSource`.

```
// Правильно
@ParameterizedTest
@ValueSource(strings = ["§ed2d", "sdf_", "123123", "§_sdf__dfww!"])
public void rejectedInvalidTokens(String invalidToken) {
    client.perform(get("/products").param("token", invalidToken))
            .andExpect(status().is(400))
}

@ParameterizedTest
@EnumSource(WorkflowState::class, mode = EnumSource.Mode.INCLUDE, names = ["FAILED
", "SUCCEEDED"])
public void dontProcessWorkflowInCaseOfAFinalState(WorkflowState itemsInitialState
) {
    // ...
}
```

Я горячо рекомендую использовать этот приём по максимуму, поскольку он позволяет тестировать больше кейсов с минимальными трудозатратами.

Наконец, я хочу обратить ваше внимание на `@CsvSource` и `@MethodSource`, которые можно использовать для более сложной параметризации, где также надо контролировать результат: вы можете передать его в одном из параметров.

```
@ParameterizedTest
@CsvSource({
    "1, 1, 2",
    "5, 3, 8",
    "10, -20, -10"
})
public void add(int summand1, int summand2, int expectedSum) {
    assertThat(calculator.add(summand1, summand2)).isEqualTo(expectedSum);
}
```

@MethodSource особенно эффективен в связке с отдельным тестовым объектом, содержащим все нужные параметры и ожидаемые результаты. К сожалению, в Java описание таких структур данных (т.н. POJO) очень громоздки. Поэтому я приведу пример с использованием дата-классов Kotlin.

```
data class TestData(
    val input: String?,
    val expected: Token?
)

@ParameterizedTest
@MethodSource("validTokenProvider")
fun `parse valid tokens`(data: TestData) {
    assertThat(parse(data.input)).isEqualTo(data.expected)
}

private fun validTokenProvider() = Stream.of(
    TestData(input = "1511443755_2", expected = Token(1511443755, "2")),
```

```
    TestData(input = "151175_13521", expected = Token(151175, "13521")),
    TestData(input = "151144375_id", expected = Token(151144375, "id")),
    TestData(input = "15114437599_1", expected = Token(15114437599, "1")),
    TestData(input = null, expected = null)
)
```

Группируйте тесты

Аннотация `@Nested` из JUnit5 удобна для группировки тестовых методов. Логически имеет смысл группировать вместе определённые типы тестов (типа `InputIsXY`, `ErrorCases`) или собрать в свою группу методы каждого теста (`GetDesign` и `UpdateDesign`).

```
public class DesignControllerTest {
    @Nested
    class GetDesigns {
        @Test
        void allFieldsAreIncluded() {}
        @Test
        void limitParameter() {}
        @Test
        void filterParameter() {}
    }
    @Nested
    class DeleteDesign {
```

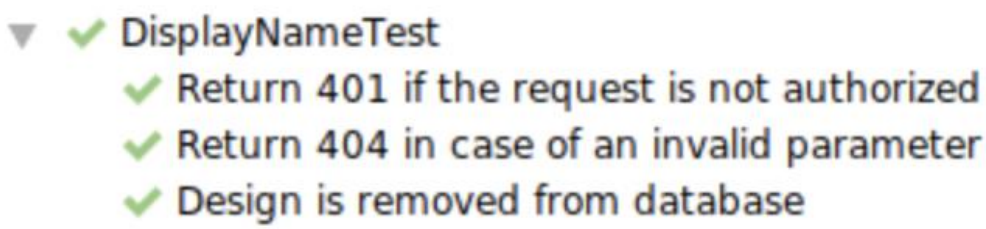
```
        @Test
        void designIsRemovedFromDb() {}
        @Test
        void return404OnInvalidIdParameter() {}
        @Test
        void return401IfNotAuthorized() {}
    }
}
```

Читаемые названия тестов при помощи @DisplayName или обратных кавычек в Kotlin

В Java можно использовать аннотацию `@DisplayName`, чтобы дать тестам более читаемые названия.

```
public class DisplayNameTest {
    @Test
```

```
    @DisplayName("Design is removed from database")
    void designIsRemoved() {}
    @Test
    @DisplayName("Return 404 in case of an invalid parameter")
    void return404() {}
    @Test
    @DisplayName("Return 401 if the request is not authorized")
    void return401() {}
}
```



В Kotlin можно использовать имена функций с пробелами внутри, если заключить их в обратные одиночные кавычки. Так вы получите читаемость результатов без избыточности кода.

```
@Test
fun `design is removed from db`() {}
```

Имитируйте внешние сервисы

Для тестирования HTTP-клиентов нам необходимо имитировать сервисы, к которым они обращаются. Я часто использую в этих целях MockWebServer из OkHttp. Альтернативами могут служить WireMock или Mockserver из Testcontainers.

```
MockWebServer serviceMock = new MockWebServer();
```

```
serviceMock.start();
HttpUrl baseUrl = serviceMock.url("/v1/");
ProductClient client = new ProductClient(baseUrl.host(), baseUrl.port());
serviceMock.enqueue(new MockResponse()
        .addHeader("Content-Type", "application/json")
        .setBody("{\"name\": \"Smartphone\"}"));
ProductDTO productDTO = client.retrieveProduct("1");
assertThat(productDTO.getName()).isEqualTo("Smartphone");
```

Используйте Awaitility для тестирования асинхронного кода

Awaitility — это библиотека для тестирования асинхронного кода. Вы можете указать, сколько раз надо повторять попытки проверки результата перед тем, как признать тест неудачным.

```
private static final ConditionFactory WAIT = await()
        .atMost(Duration.ofSeconds(6))
        .pollInterval(Duration.ofSeconds(1))
        .pollDelay(Duration.ofSeconds(1));
@Test
public void waitAndPoll(){
    triggerAsyncEvent();
    WAIT.untilAsserted(() -> {
```

```
        assertThat(findInDatabase(1).getState()).isEqualTo(State.SUCCESS);
    });
}
```

Не надо резолвить DI-зависимости (Spring)

Инициализация DI-фреймворка занимает несколько секунд перед тем, как тесты могут стартовать. Это замедляет цикл обратной связи, особенно на начальном этапе разработки.

Поэтому я стараюсь не использовать DI в интеграционных тестах, а создаю нужные объекты вручную и «провязываю» их между собой. Если вы используете внедрение в конструктор, то это самое простое. Как правило, в своих тестах вы проверяете бизнес-логику, а для этого DI не нужно.

Более того, начиная с версии 2.2, Spring Boot поддерживает ленивую инициализацию бинов, что заметно ускоряет тесты, использующие DI.

Ваш код должен быть тестируемым

Не используйте статический доступ. Никогда

Статический доступ — это антипаттерн. Во-первых, он запутывает зависимости и побочные эффекты, делая весь код сложночитаемым и подверженным неочевидным ошибкам. Во-вторых, статический доступ мешает тестированию. Вы больше не можете заменять объекты, но в тестах вам нужно использовать моки или реальные объекты с другой конфигурацией (например, DAO-объект, указывающий на тестовую базу данных).

Вместо статического доступа к коду положите его в нестатический метод, создайте экземпляр класса и передайте полученный объект в конструктор.

```
// Неправильно
public class ProductController {
    public List<ProductDTO> getProducts() {
        List<ProductEntity> products = ProductDAO.getProducts();
        return mapToDTOs(products);
    }
}
```

```
// Правильно
public class ProductController {
    private ProductDAO dao;
    public ProductController(ProductDAO dao) {
        this.dao = dao;
    }
    public List<ProductDTO> getProducts() {
        List<ProductEntity> products = dao.getProducts();
        return mapToDTOs(products);
    }
}
```

К счастью, DI-фреймворки типа Spring предоставляют инструменты, делающие статический доступ ненужным, автоматически создавая и связывая объекты без нашего участия.

Параметризуйте

Все релевантные части класса должны иметь возможность настройки со стороны теста. Такие настройки можно передавать в конструктор класса.

Представьте, например, что ваш DAO имеет фиксированный лимит в 1000 объектов на запрос. Чтобы проверить этот лимит, вам надо будет перед тестом добавить в тестовую БД 1001 объект. Используя аргумент конструктора, вы можете сделать это значение настраиваемым: в продакшене оставить 1000, в тестировании сократить до 2. Таким образом, чтобы проверить работу лимита вам будет достаточно добавить в тестовую БД всего 3 записи.

Используйте внедрение в конструктор

Внедрение полей — зло, оно ведёт к плохой тестируемости кода. Вам необходимо инициализировать DI перед тестами или заниматься стрёмной магией рефлексий. Поэтому предпочтительно использовать внедрение через конструктор, чтобы легко контролировать зависимые объекты в процессе тестирования.

На Java придётся написать немного лишнего кода:

```
// Правильно
public class ProductController {
    private ProductDAO dao;
    private TaxClient client;

    public ProductController(ProductDAO dao, TaxClient client) {
        this.dao = dao;
        this.client = client;
    }
}
```

В Kotlin тоже самое пишется намного лаконичнее:

```
// Правильно
class ProductController(
    private val dao: ProductDAO,
    private val client: TaxClient
){
}
```

Не используйте `Instant.now()` или `new Date()`

Не надо получать текущее время вызовами `Instant.now()` или `new Date()` в продакшн-коде, если вы хотите тестировать это поведение.

```
// Неправильно
public class ProductDAO {
    public void updateDateModified(String productId) {
        Instant now = Instant.now(); // !
        Update update = Update()
            .set("dateModified", now);
        Query query = Query()
            .addCriteria(where("_id").eq(productId));
        return mongoTemplate.updateOne(query, update, ProductEntity.class);
    }
}
```

Проблема в том, что полученное время не может контролироваться со стороны теста. Вы не сможете сравнить полученный результат с конкретным значением, потому что он всё время разный. Вместо этого используйте класс `Clock` из Java.

```
// Правильно
public class ProductDAO {
    private Clock clock;

    public ProductDAO(Clock clock) {
        this.clock = clock;
    }

    public void updateProductState(String productId, State state) {
        Instant now = clock.instant();
```

```
        // ...
    }
}
```

В этом тесте вы можете создать мок-объект для `Clock`, передать его в `ProductDAO` и сконфигурировать мок-объект так, чтобы он возвращал одно и то же время. После вызовы `updateProductState()` мы сможем проверить, что в базу данных попало именно заданное нами значение.

Разделяйте асинхронное выполнение и собственно логику

Тестирование асинхронного кода — непростая штука. Библиотеки типа Awaitility оказывают большую помощь, но процесс всё равно запутан, и мы можем получить «моргающий» тест. Есть смысл разделять бизнес-логику (обычно синхронную) и асинхронный инфраструктурный код, если такая возможность имеется.

Например, вынеся бизнес-логику в ProductController мы сможем запросто протестировать её синхронно. Вся асинхронная и параллельная логика останутся в ProductScheduler, который можно протестировать изолированно.

```
// Правильно
public class ProductScheduler {
    private ProductController controller;
    @Scheduled
    public void start() {
        CompletableFuture<String> usFuture = CompletableFuture.supplyAsync(() -> c
ontroller.doBusinessLogic(Locale.US));
        CompletableFuture<String> germanyFuture = CompletableFuture.supplyAsync(()
-> controller.doBusinessLogic(Locale.GERMANY));
        String usResult = usFuture.get();
        String germanyResult = germanyFuture.get();
```

```
    }
}
```