Gradle HOW-TO

Contents

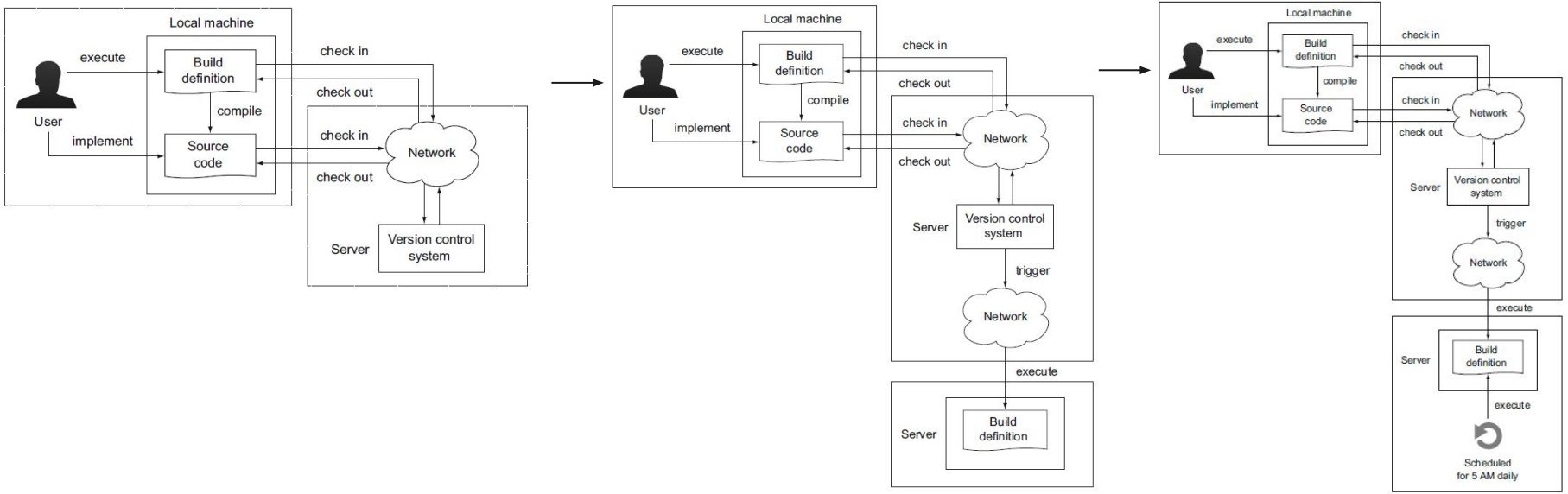
Introduction to project automatization

With project automatization vs Without project automatization

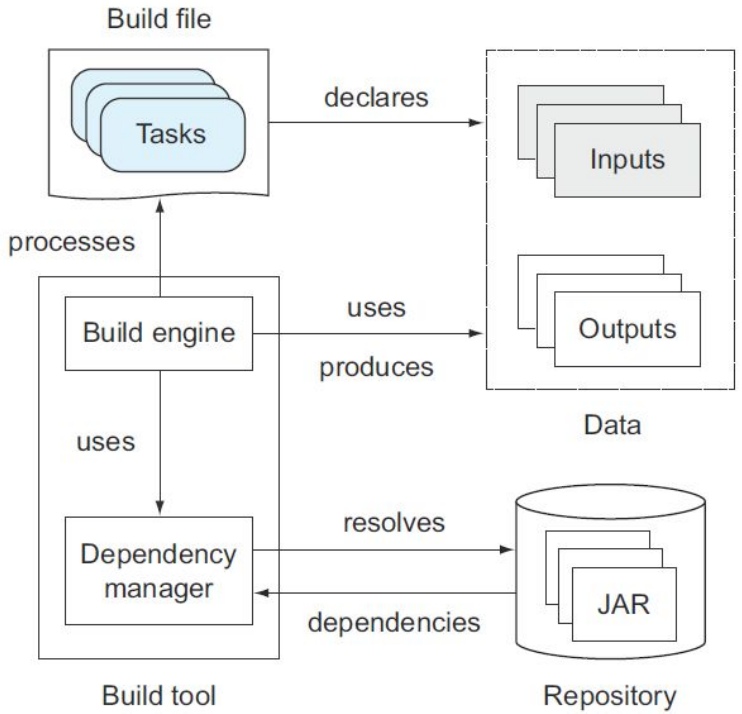
- My IDE does the job
- It works on my box.
- The code integration is a complete disaster.
- The testing process slows to a crawl.
- Deployment turns into a marathon.

- Prevents manual intervention
- Creates repeatable builds
- Makes builds portable

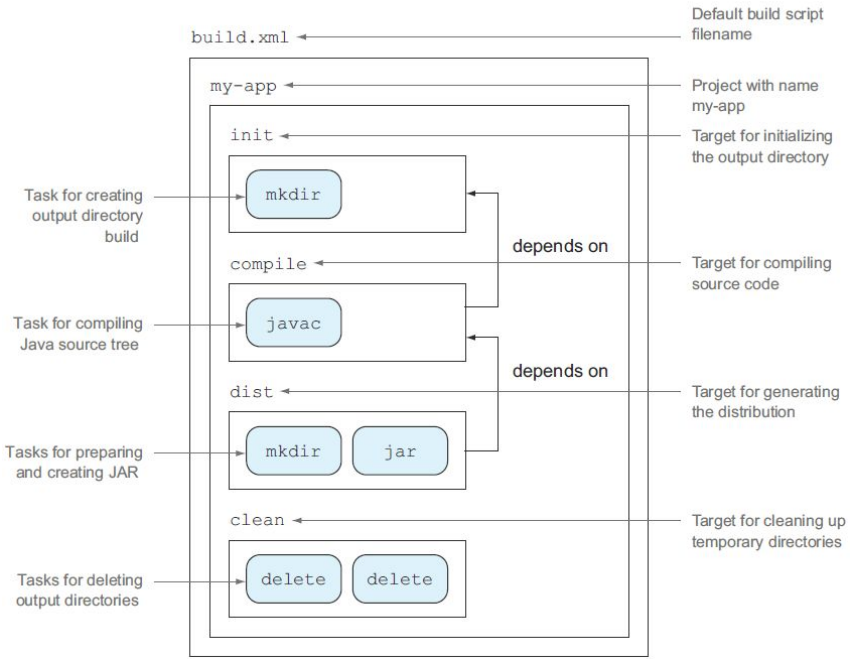
Types of project automatization



Anatomy of build tools



Apache Ant




```
<project name="my-app" default="dist" basedir=".">
   <property name="src" location="src"/>
   <property name="build" location="build"/>
   <property name="dist" location="dist"/>
   <property name="version" value="1.0"/>

   <target name="init">
      <mkdir dir="${build}"/>
   </target>

   <target name="compile" depends="init" description="compile the source">
      <javac srcdir="${src}" destdir="${build}"
          ➥ classpath="lib/commons-lang3-3.1.jar"
          ➥ includeantruntime="false"/>
   </target>

   <target name="dist" depends="compile"
        ➥ description="generate the distribution">
      <mkdir dir="${dist}"/>
      <jar jarfile="${dist}/my-app-${version}.jar" basedir="${build}"/>
   </target>

   <target name="clean" description="clean up">
      <delete dir="${build}"/>
      <delete dir="${dist}"/>
   </target>
</project>
```

Sets global properties for this build, like source, output, and distribution directories

Creates build directory structure used by compile target

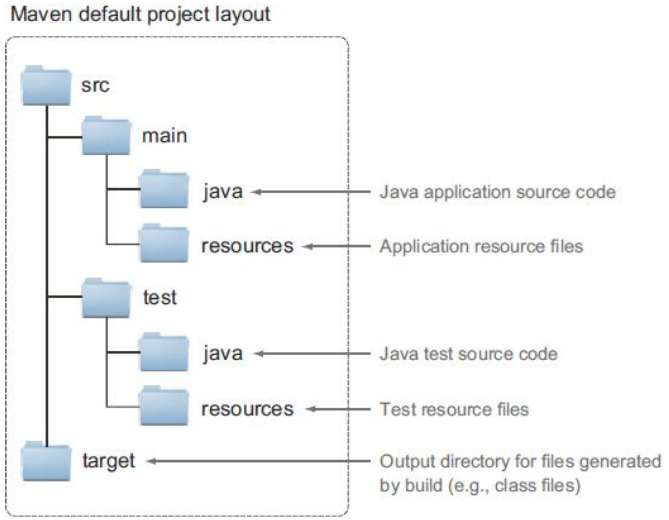
Compiles Java code from directory src into directory build

Creates distribution directory

Assembles everything in directory build into JAR file myapp-l.0

Deletes build and dist directory trees

Apache Maven



Project definition including referenced XML schema to validate correct structure and content of document.

Name of project that automatically determines name of produced artifact (in this case the JAR file).

```
<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.mycompany.app</groupId>
   <artifactId>my-app</artifactId>
   <packaging>jar</packaging>
   <version>1.0</version>

   <dependencies>
      <dependency>
         <groupId>org.apache.commons</groupId>
         <artifactId>commons-lang3</artifactId>
         <version>3.1</version>
         <scope>compile</scope>
      </dependency>
   </dependencies>
</project>
```

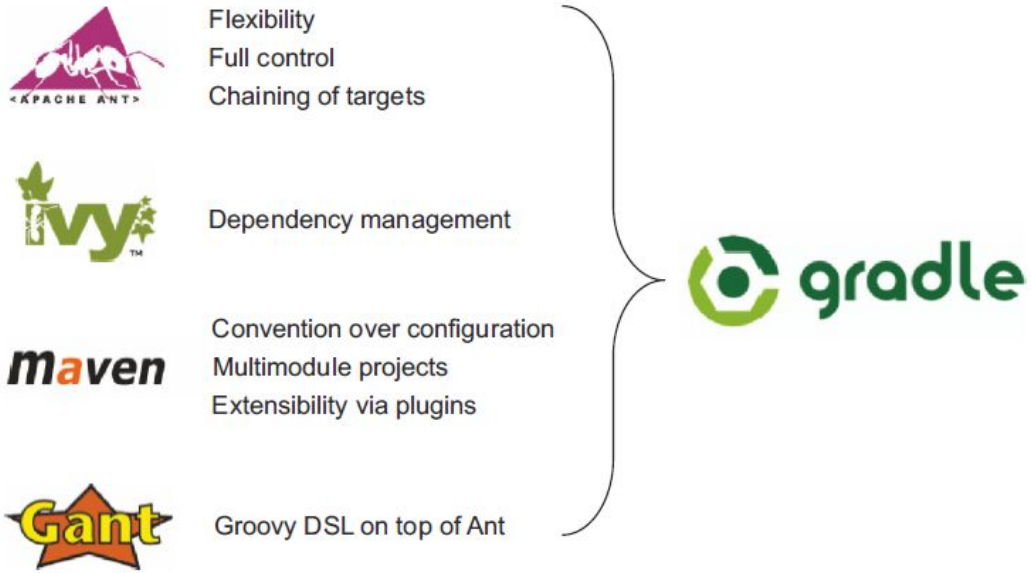
Version of Maven's internal model.

Identifies the organization the project belongs to.

Type of artifact produced by project.

Version of project that factors into produced artifact name.

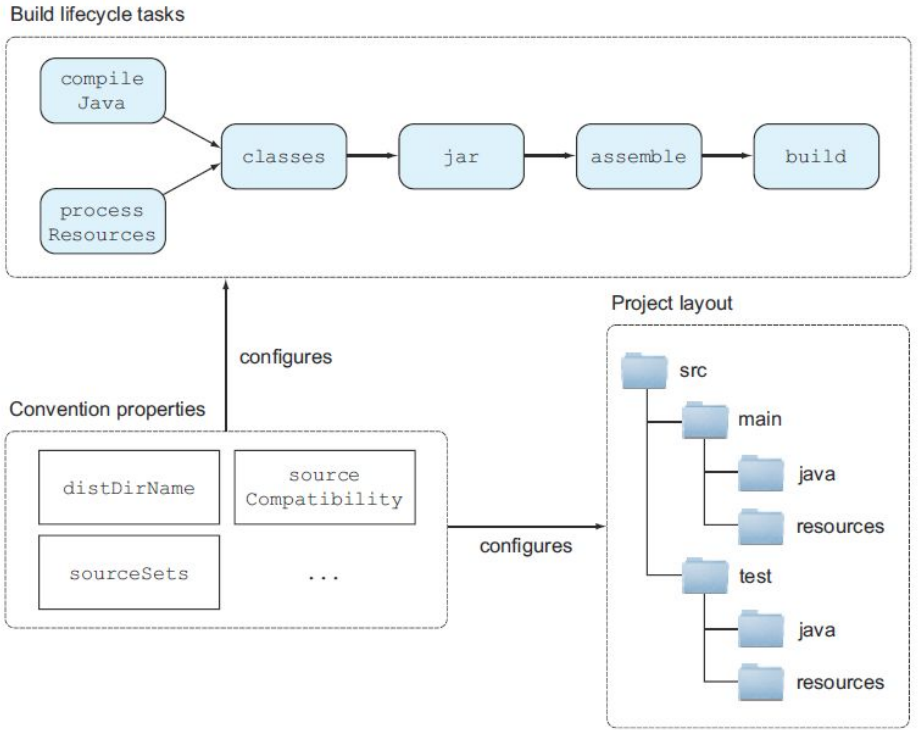
Declared dependency on Apache Commons Lang library with version 3.1; scope of a dependency determines lifecycle phase it's applied to. In this case it's needed during compilation phase.


Flexibility
Full control
Chaining of targets
Dependency management
Convention over configuration
Multimodule projects
Extensibility via plugins
Groovy DSL on top of Ant
gradle

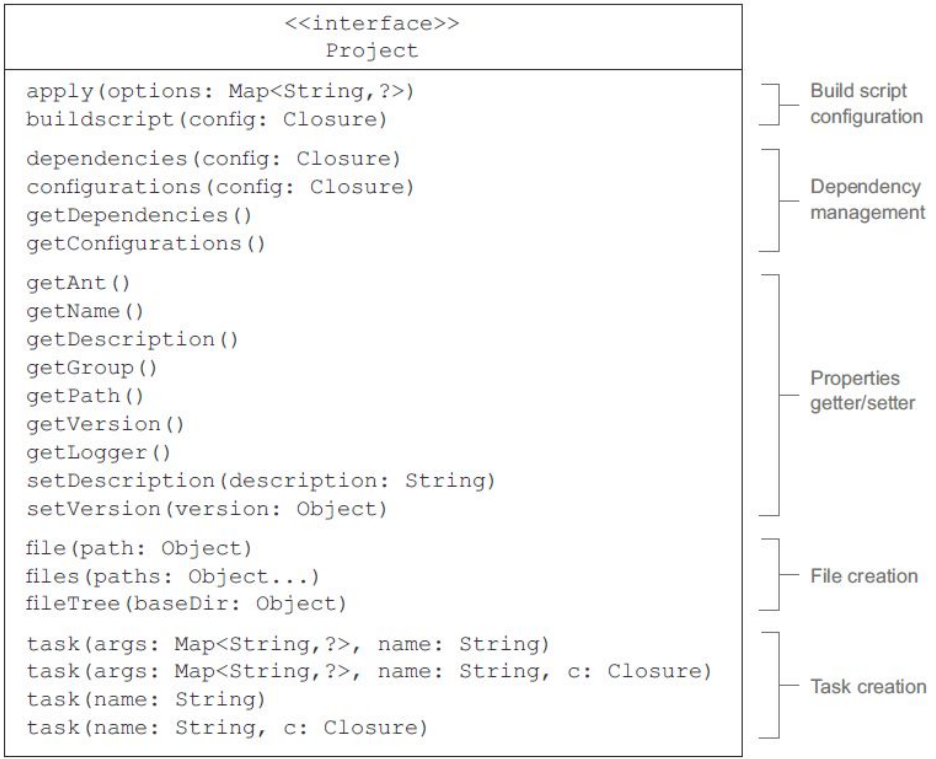
Gradle HOW-TO

Simplest configuration

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   testCompile 'junit:junit:4.11'
}
```

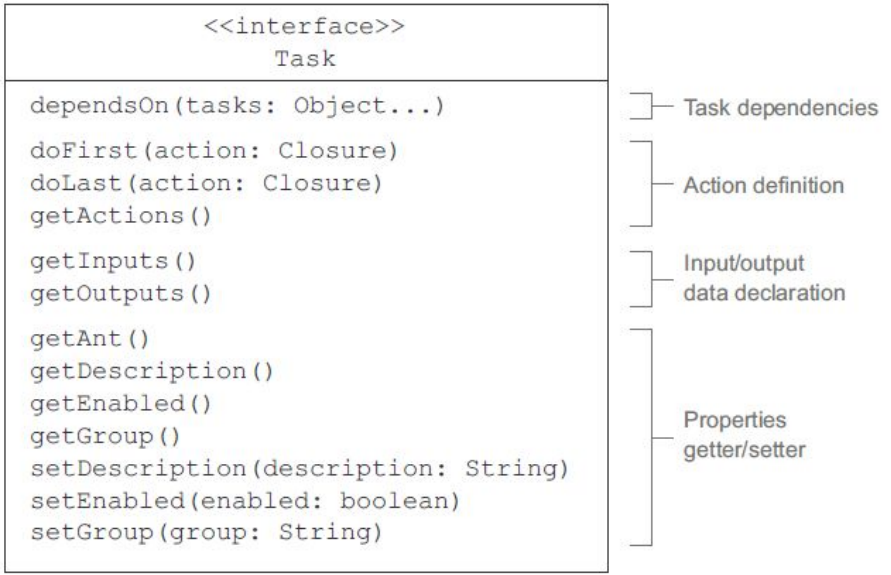


Project in Gradle



Task

```
version = '0.1-SNAPSHOT'

task printVersion {
   doLast {
      println "Version: $version"
   }
}
```

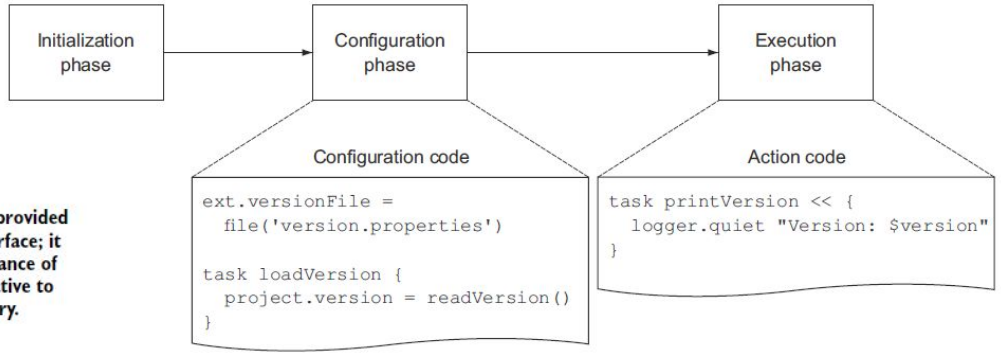



```
task first << { println "first" }
task second << { println "second" }

task printVersion(dependsOn: [second, first]) << {
   logger.quiet "Version: $version"
}

task third << { println "third" }
third.dependsOn('printVersion')
```

Assigning multiple task dependencies

Referencing task by name when declaring dependency

Task configuration




```
ext.versionFile = file('version.properties')

task loadVersion {
   project.version = readVersion()
}

ProjectVersion readVersion() {
   logger.quiet 'Reading the version file.'

   if(!versionFile.exists()) {
      throw new GradleException("Required version file does not exist:
         ➥ $versionFile.canonicalPath")
   }

   Properties versionProps = new Properties()

   versionFile.withInputStream { stream ->
      versionProps.load(stream)
   }

   new ProjectVersion(versionProps.major.toInteger(),
   ➥ versionProps.minor.toInteger(), versionProps.release.toBoolean())
}
```

File method is provided by Project interface; it creates an instance of java.io.File relative to project directory.

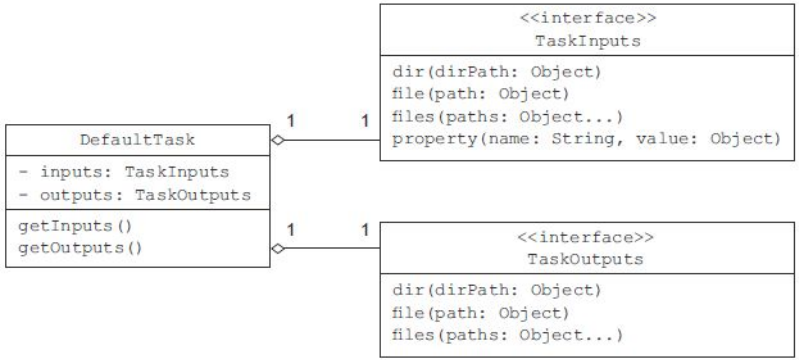
Task configuration is defined without left shift operator.

If version file doesn't exist throw a GradleException with an appropriate error message.

Groovy's file implementation adds methods to read it with newly created InputStream.

In Groovy you can omit the return keyword if it's last statement in method.

Declaring task in-out




```
task makeReleaseVersion(group: 'versioning', description: 'Makes project
                        ➥ a release version.') {
    inputs.property('release', version.release)
    outputs.file versionFile

    doLast {
        version.release = true
        ant.propertyfile(file: versionFile) {
            entry(key: 'release', type: 'string', operation: '=', value: 'true')
        }
    }
}
```

Inputs/ outputs are declared during configuration phase

As the version file is going to be modified it's declared as output file property

Declaring version release property as input

```
$ gradle makeReleaseVersion
:makeReleaseVersion

$ gradle makeReleaseVersion
:makeReleaseVersion UP-TO-DATE
```

Custom Task Class

```
class ReleaseVersionTask extends DefaultTask {
   @Input Boolean release
   @OutputFile File destFile

   ReleaseVersionTask() {
      group = 'versioning'
      description = 'Makes project a release version.'
   }

   @TaskAction
   void start() {
      project.version.release = true
      ant.propertyfile(file: destFile) {
         entry(key: 'release', type: 'string', operation: '=', value: 'true')
      }
   }
}
```

Declaring custom task's inputs/ outputs through annotations

Setting task's group and description properties in the constructor

Annotation declares method to be executed

Writing a custom task that extends Gradle's default task implementation

```
task makeReleaseVersion(type: ReleaseVersionTask) {
   release = version.release
   destFile = versionFile
}
```

Setting custom task properties

Defining an enhanced task of type ReleaseVersionTask

Task Rule

```
task incrementMajorVersion(group: 'versioning', description: 'Increments
                        ➥ project major version.') << {
   String currentVersion = version.toString()
   ++version.major
   String newVersion = version.toString()
   logger.info "Incrementing major project version: $currentVersion ->
               ➥ $newVersion"

   ant.propertyfile(file: versionFile) {
      entry(key: 'major', type: 'int', operation: '+', value: 1)
   }
}

task incrementMinorVersion(group: 'versioning', description: 'Increments
                        ➥ project minor version.') << {
   String currentVersion = version.toString()
   ++version.minor
   String newVersion = version.toString()

   logger.info "Incrementing minor project version: $currentVersion ->
               ➥ $newVersion"

   ant.propertyfile(file: versionFile) {
      entry(key: 'minor', type: 'int', operation: '+', value: 1)
   }
}
```

Using Ant task propertyfile to increment a specific property within a property file

Using Ant task propertyfile to increment a specific property within a property file

```
tasks.addRule("Pattern: increment<Classifier>Version - Increments the
            ➥ project version classifier.") { String taskName ->
   if(taskName.startsWith('increment') && taskName.endsWith('Version')) {
      task(taskName) << {
         String classifier = (taskName - 'increment' - 'Version')
                           ➥ .toLowerCase()
         String currentVersion = version.toString()

         switch(classifier) {
            case 'major': ++version.major
                          break
            case 'minor': ++version.minor
                          break
            default: throw new GradleException("Invalid version
                     ➥ type '$classifier. Allowed types: ['Major', 'Minor']")
         }

         String newVersion = version.toString()
         logger.info "Incrementing $classifier project version:
                     ➥ $currentVersion -> $newVersion"

         ant.propertyfile(file: versionFile) {
            entry(key: classifier, type: 'int', operation: '+', value: 1)
         }
      }
   }
}
```

Adding a task rule with provided description

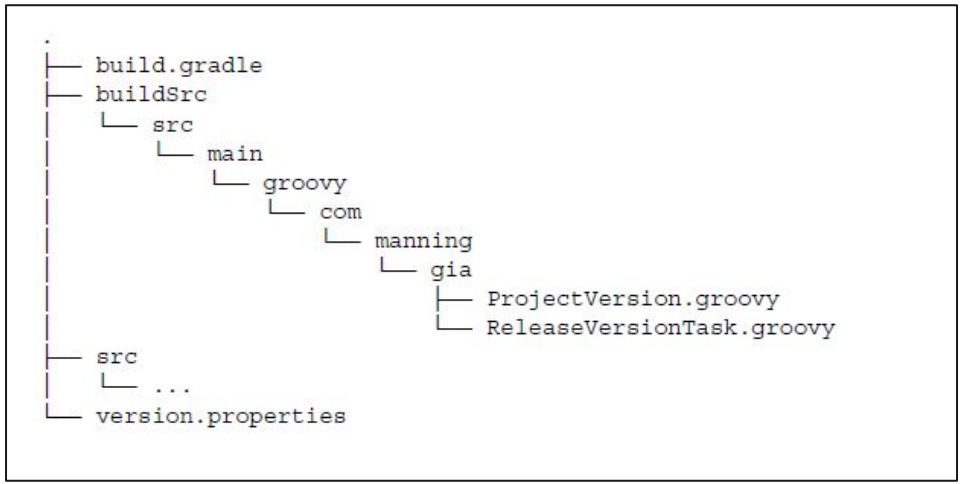
Extracting type string from full task name

Dynamically add a task named after provided pattern with a doLast action

Checking task name for predefined pattern

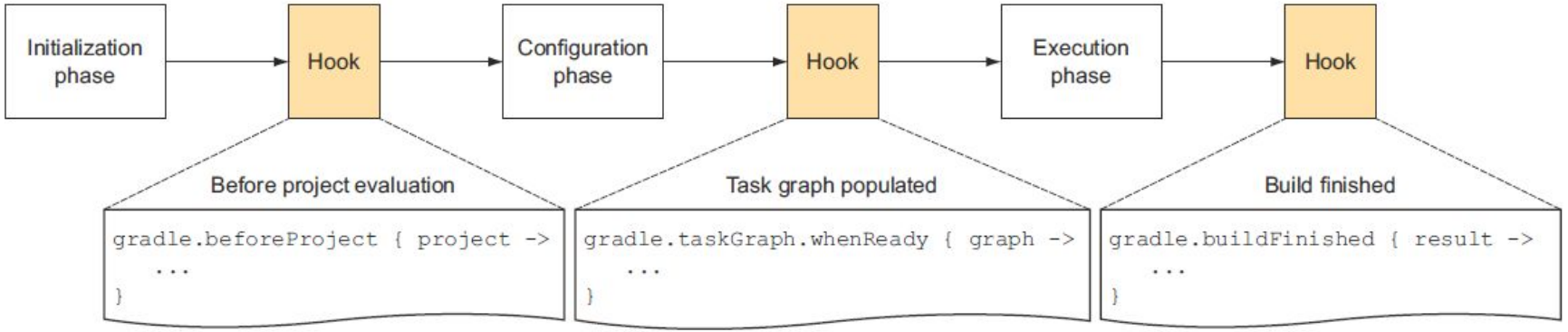
```
$ gradle tasks
...
Rules
-----
Pattern: increment<Classifier>Version - Increments project version type
```

Building code in buildSrc directory

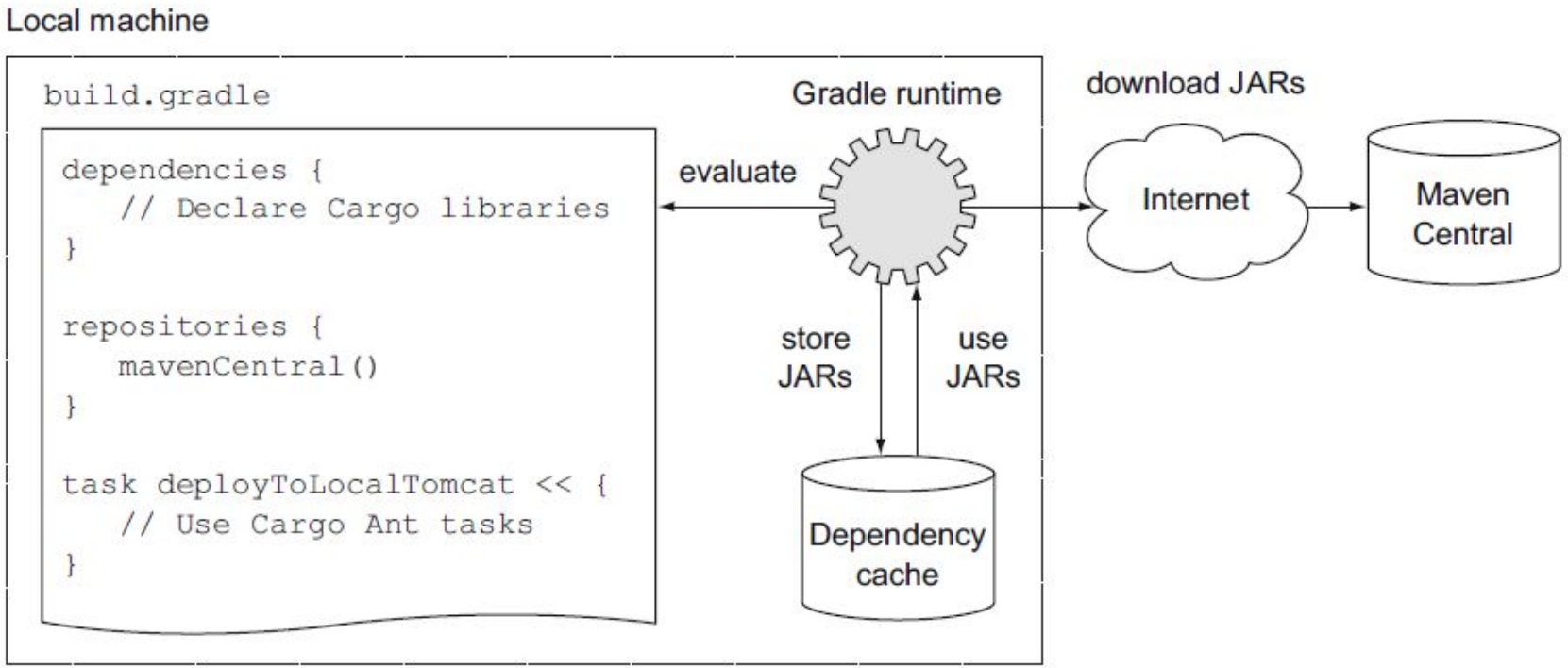



```
$ gradle makeReleaseVersion
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test
:buildSrc:check
:buildSrc:build
:makeReleaseVersion UP-TO-DATE
```

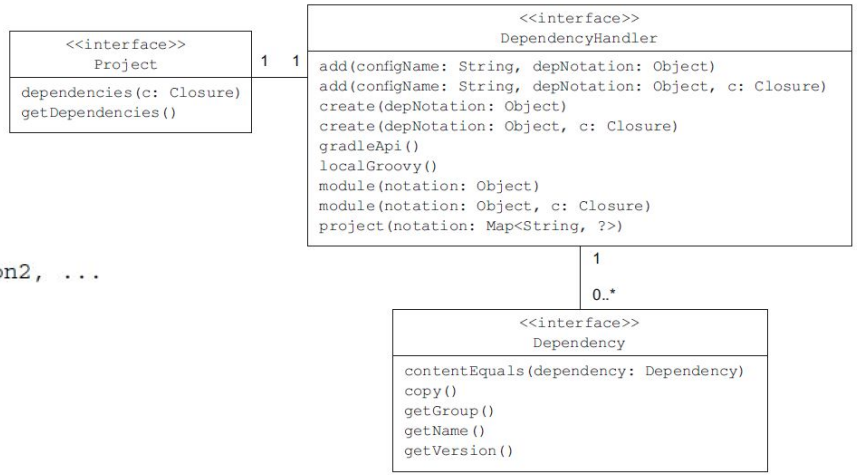
Hooking into the build lifecycle



Dependency management



Dependencies management in Gradle




```
dependencies {
   configurationName dependencyNotation1, dependencyNotation2, ...
}
```

```
org.hibernate:hibernate-core:3.6.3-Final
└──────────┘ └─────────────┘ └──────────┘
     group          name         version
```

File dependencies

```
task copyDependenciesToLocalDir(type: Copy) {
   from configurations.cargo.asFileTree
   into "${System.properties['user.home']}/libs/cargo"
}
```

Syntactic sugar provided by Gradle API; same as calling configurations.getByName-('cargo').asFileTree.

```
dependencies {
   cargo fileTree(dir: "${System.properties['user.home']}/libs/cargo",
                  ➥ include: '*.jar')
}
```

Maven repo configuration

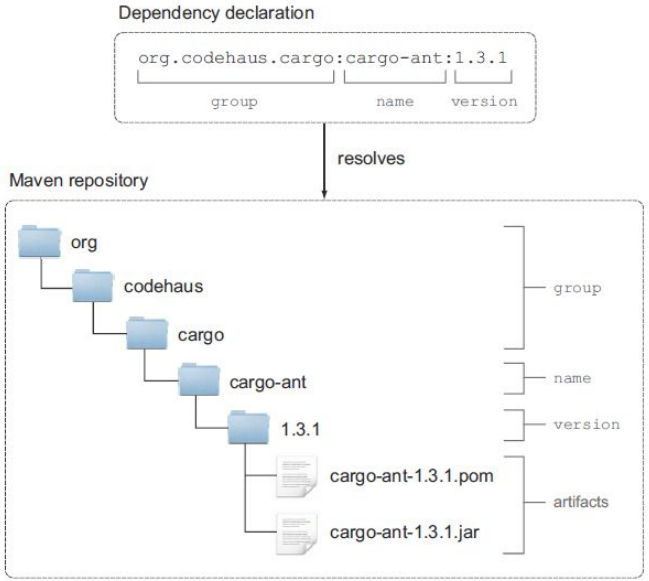
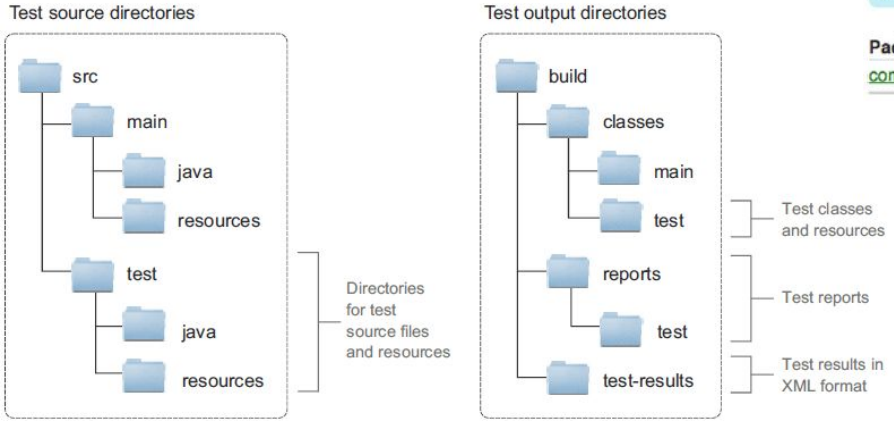
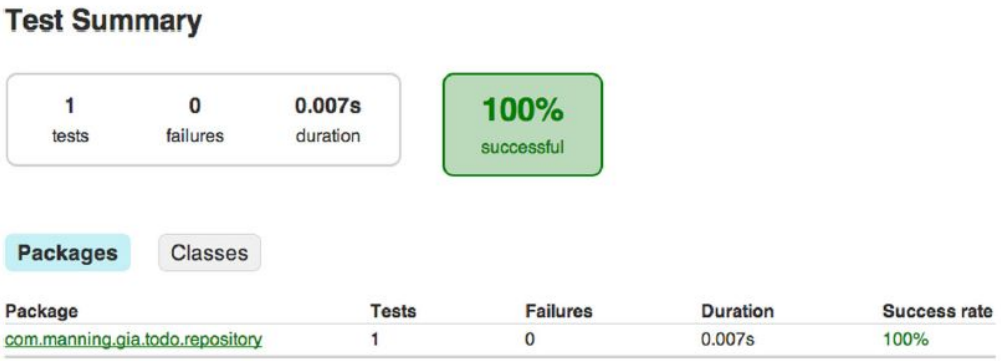


Figure 5.9 How a dependency declaration maps to artifacts in a Maven repository

```
repositories {
   mavenCentral()
   maven {
      name 'Custom Maven Repository',
      url 'http://repository-gradle-in-action.forge.cloudbees.com/release/')
   }
}
```

Testing with Gradle

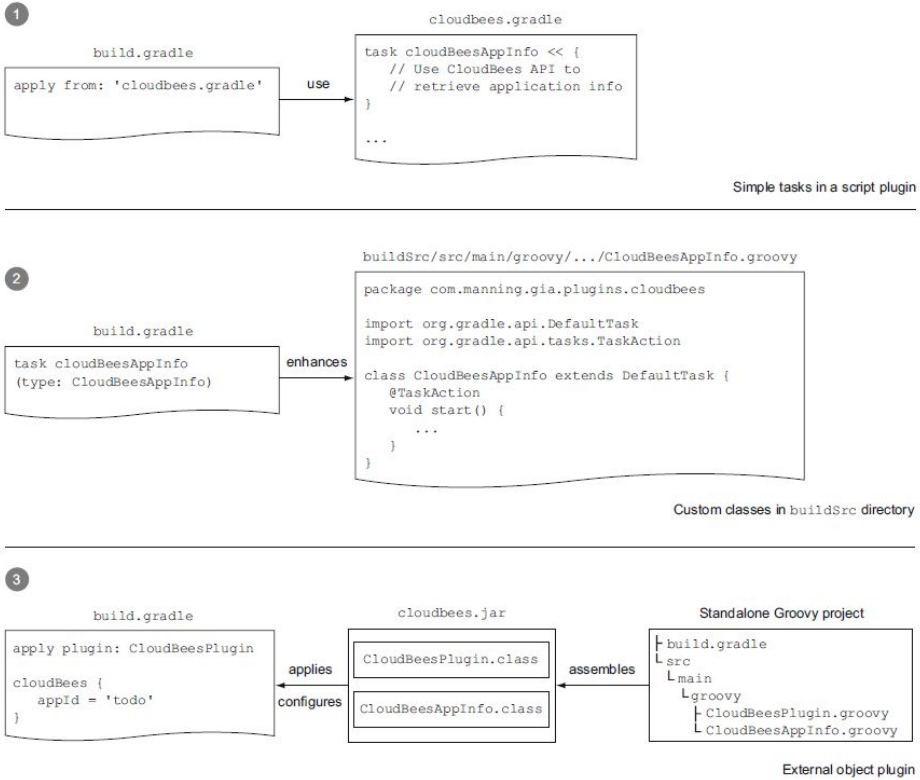





```
dependencies {
   testCompile 'junit:junit:4.11'
}
```

PyCharm plugin

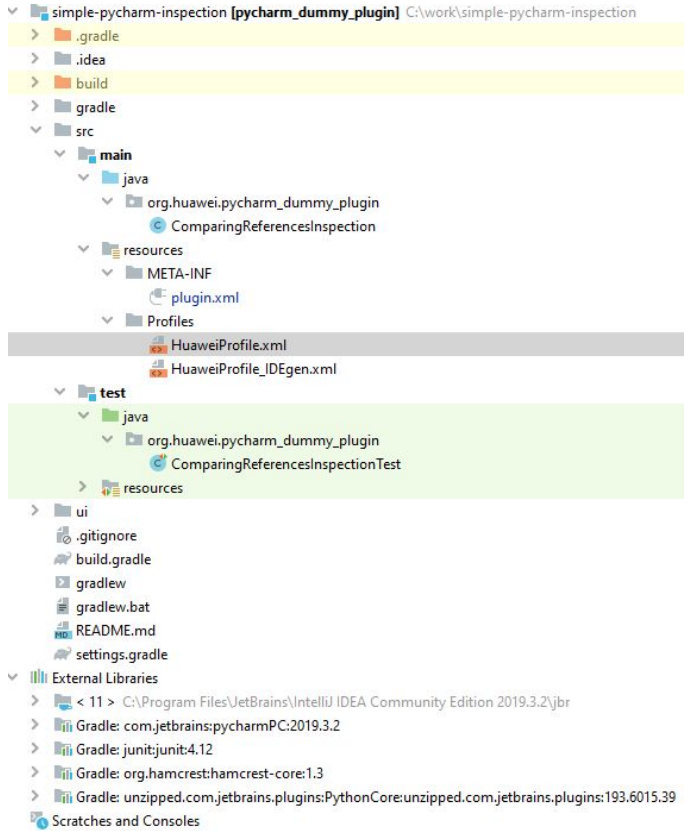
Usual cases of gradle plugins



Gradle intellij plugin

Intellij tasks

- **buildPlugin** - Bundles the project as a distribution.
- **buildSearchableOptions** - Builds searchable options for plugin.
- **jarSearchableOptions** - Jars searchable options.
- **patchPluginXml** - Patch plugin xml files with corresponding since/until build numbers and version attributes
- **prepareSandbox** - Prepare sandbox directory with installed plugin and its dependencies.
- **prepareTestingSandbox** - Prepare sandbox directory with installed plugin and its dependencies.
- **publishPlugin** - Publish plugin distribution on plugins.jetbrains.com.
- **runIde** - Runs Intellij IDEA with installed plugin.
- **verifyPlugin** - Validates completeness and contents of plugin.xml descriptors as well as plugin's archive structure.



Some actions for a start (build.gradle)

```
plugins {
    id 'java'
    id 'org.jetbrains.intellij' version '0.4.16'
}
```

```
test {
    // Set idea.home.path to the absolute path to the intellij-community source
    // on your local machine.
    systemProperty "idea.home.path", 'C:\\work\\test_pycharm\\PyCharm Community Edition 2019.3.2'
}
```

```
intellij {
    // Define IntelliJ Platform against which to build the plugin project.
    // Use the IntelliJ Platform BRANCH.BUILD version matching "targetIDE" (PhpStorm)
    version 'PC-2019.3.2'
    type 'PC'
    updateSinceUntilBuild false
    plugins 'PythonCore:193.6015.39'
    downloadSources = true

}

runIde {
    // Absolute path to the installed targetIDE to use as IDE Development Instance
    // Note the Contents directory must be added at the end of the path for macOS.
    ideDirectory 'C:\\work\\test_pycharm\\PyCharm Community Edition 2019.3.2'
}
```

Some actions for a start (plugin.xml)

```
<idea-plugin>
    <id>org.huawei.pycharm_dummy_plugin</id>
    <name>Huawei dummy inspection</name>
    <vendor email="support@yourcompany.com" url="http://www.yourcompany.com">YourCompany</vendor>

    <!-- please see http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/plugin_compatibility.html
         on how to target different products -->
    <depends>com.intellij.modules.python</depends>
    <depends>PythonCore</depends>

    <extensions defaultExtensionNs="com.intellij">
        <!-- Add your extensions here -->
        <localInspection language="Python" shortName="ComparingReferencesInspection" suppressId="PyTypedDict"
                         displayName="Huawei the best company in the world" groupKey="INSP.GROUP.python"
                         enabledByDefault="true"  level="WARNING"
                         implementationClass="org.huawei.pycharm_dummy_plugin.ComparingReferencesInspection"/>
    </extensions>

    <actions>
        <!-- Add your actions here -->
    </actions>
</idea-plugin>
```

Inspection Description(HuaweiProfile.xml)

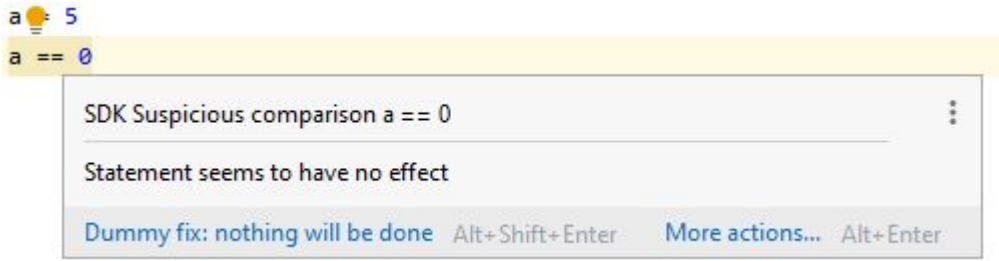
```
<profile version="1.0">
  <option name="myName" value="HuaweiInspections" />
  <inspection_tool class="ComparingReferencesInspection" enabled="true" level="WARNING" enabled_by_default="true" />
</profile>
```

Our dummy Inspection

- Describing an inspection in the plugin configuration file.
- Implementing a local inspection class to inspect Python code in the IntelliJ Platform-based IDE editor.
- Creating a visitor to traverse the `PsiTree` of the Pyton file being edited, inspecting for problematic syntax.
- Implementing a quick fix class to correct syntax problems by altering the `PsiTree` as needed. Quick fixes are displayed to the user like intentions.
- Implementing an inspection preferences panel to display information about the inspection.
- Writing an HTML description of the inspection for display in the inspection preferences panel.
- Optionally, create a unit test for the plugin.

How does it looks in PyCharm

- From IDE



- From console.
  (Result is stored in xml)

```
<problems is_local_tool="true">
<problem>
  <file>file://$PROJECT_DIR$/atf/atf_error.py</file>
  <line>4</line>
  <module>analyzer-benchmark</module>
  <entry_point TYPE="file" FQNAME="file://$PROJECT_DIR$/atf/atf_error.py" />
  <problem_class severity="WARNING" attribute_key="WARNING_ATTRIBUTES">Huawei the best company in the world</problem_class>
  <description>SDK Suspicious comparison a==0 #loc</description>
  <highlighted_element>a==0</highlighted_element>
  <offset>3</offset>
  <length>4</length>
</problem>
</problems>
```